НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
Завідувач кафедри
_____ Сергій СТІРЕНКО
«___»_____20__ р.

Дипломний проект

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Комп'ютерні системи та мережі»

спеціальності 123 «Комп'ютерна інженерія»

на тему: «Комп'ютерна гра в жанрі Аркада»

Виконав:
студент IV курсу, групи ІО-62
Черпатюк Андрій Сергійович _____

Керівник:
Доцент, кандидат технічних наук,
Верба Олександр Андрійович _____

Консультант з нормоконтролю:
Професор, доктор технічних наук
Сімоненко Валерій Павлович _____

Рецензент:
Доцент, кандидат технічних наук
Юрчишин Василь Якович _____

Засвідчую, що у цьому дипломному проекті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2020 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 123 «Комп'ютерна інженерія»

Освітньо-професійна програма «Комп'ютерні системи та мережі»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Сергій СТІРЕНКО

«___»_____20__ р.

ЗАВДАННЯ

на дипломний проект студенту

Черпатюку Андрію Сергійовичу

1. Тема проекту «Комп'ютерна гра в жанрі Аркада», керівник проекту Верба Олександр Андрійович, доцент, к.т.н., затверджені наказом по університету від «07» травня 2020 р. № 1081-с

2. Термін подання студентом проекту 06 червня 2020 р.

3. Вихідні дані до проекту: технічне завдання, науково-технічна література

4. Зміст пояснювальної записки: порівняльний аналіз існуючих програмних рішень, вибір засобів реалізації та опис отриманої системи

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслеників, плакатів, презентацій тощо) :

1. Схема класів – плакат
2. Алгоритм роботи програми – плакат
3. Схема взаємодії з користувачем – плакат

6. Консультанти розділів проекту

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Сімоненко В.П., професор		

7. Дата видачі завдання 01 вересня 2019 р.

Календарний план

№ з/п	Назва етапів виконання дипломного проєкту	Термін виконання етапів проєкту	Примітка
1	Затвердження теми роботи	01.09.2019-22.12.2019	
2	Вивчення та аналіз завдання	23.12.2019-20.03.2020	
3	Розробка архітектури та загальної структури системи	20.03.2020-01.04.2020	
4	Розробка структур окремих підсистем	01.04.2020-10.04.2020	
5	Програмна реалізація системи	11.04.2020-20.04.2020	
6	Оформлення пояснювальної записки	01.05.2020-23.05.2020	
7	Передзахист	24.05.2020-26.05.2020	
8	Захист	15.06.2020-20.06.2020	

Студент Андрій ЧЕРПАТЮК

Керівник Олександр ВЕРБА

АНОТАЦІЯ

Дана дипломна робота присвячена розробці комп’ютерної гри у жанрі Аркада з використанням технологій існуючого у відкритому доступі ігрового рушія, що призначена для проведення дозвілля.

Гравець має під контролем особистий зореліт, за допомогою якого може досліджувати навколишній ігровий простір та його правила.

Для реалізації гри використовується ігровий рушій Godot. Для опису ігрової логіки була обрана мова програмування gdscript.

ANNOTATION

This thesis is devoted to the development of a computer game in the genre of Arcade using the technologies of the existing open source game engine, which is designed for entertainment.

The player has control of a personal starship, with which he can explore the surrounding in-game space and its rules.

Godot game engine is used to implement the core game. The gdscript programming language was chosen to describe the game logic.

ВІДОМІСТЬ ДИПЛОМНОГО ПРОЄКТУ

№ з/п	Формат	Позначення	Найменування	Кількість листів	Примітка
1	А4		Завдання на дипломний проект	2	
2	А4	*ІАЛЦ. 467800.001* ВП	Відомість проекту	1	
3	А4	*ІАЛЦ. 467800.002* ТЗ	Технічне завдання	3	
4	А4	*ІАЛЦ. 467800.003* ПЗ	Пояснювальна записка	56	
5	А4	*ІАЛЦ. 467800.004* Д1	Схема класів	1	
6	А4	*ІАЛЦ. 467800.005* Д2	Алгоритм роботи програми	1	
7	А4	*ІАЛЦ. 467800.006* Д3	Схема взаємодії з користувачем	1	
8	А4	*ІАЛЦ. 467800.007* Д4	Текст програми	11	

				ІАЛЦ. 467800.001 ВП		
	ПІБ	Підп.	Дата			
Розробн.	Черпатюк А.С.			Відомість дипломного проекту	Лист	Листів
Керівн.	Верба О.А.				1	1
Консульт.					КПІ ім. Ігоря Сікорського Каф. ОТ Гр. ІО-62	
Н/контр.	Сімоненко В.П.					
Зав.каф.	Стіренко С.Г.					

Технічне завдання

до дипломного проекту

на тему: «Комп’ютерна гра в жанрі Аркада»

Київ – 2020 року

ЗМІСТ

1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання розповсюджується на розробку програмного додатку на тему «Комп’ютерна гра в жанрі Аркада». Область застосування: організація дозвілля гравця.

2. ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки служить завдання на виконання роботи кваліфікаційно-освітнього рівня «бакалавр комп'ютерної інженерії», затверджене кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний інститут ім. Ігоря Сікорського».

3. МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою даного проекту є розробка ігрового додатку, що слугує для приємного проводження вільного часу відпочинку.

4. ДЖЕРЕЛА РОЗРОБКИ

Джерелами розробки є технічна документація для інтерфейсів прикладного програмування та бібліотек, що використовуються під час розробки програмного продукту, науково-технічна література з інформаційних технологій, публікації в періодичних виданнях, а також відповідні статті в мережі Інтернет за даним питанням.

5. ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до програмного продукту, що розробляється

- Платформонезалежність
- Зберігання даних на пристрої.
- Конкурентоспроможність ігрового додатку на ринку комп’ютерних ігор.
- Простий і чіткий програмний інтерфейс застосунку.

5.2. Вимоги до програмного забезпечення

- Мова програмування gdscript / Golang.
- Використання ігрового рушія Godot.

5.3. Вимоги до апаратного забезпечення

- Комп’ютер на базі процесору Intel Pentium 4 / Athlon 64 і вище.
- Оперативної пам’яті не менше 2048 Мбайт.
- 500 Мбайт вільного місця на пристрої зберігання інформації.

Пояснювальна записка

до дипломного проекту

на тему: «Комп’ютерна гра в жанрі Аркада»

Київ – 2020 року

ЗМІСТ

ПЕРЕЛІК ТЕРМІНІВ ТА СКОРОЧЕНЬ

ОС	Операційна система
ПК	Персональний комп’ютер
RTS	(англ. Real Time Strategy) стратегічна відеогра у реальному часі – жанр комп’ютерної гри
ECS	(англ. Entity-Component-System) шаблон проектування, який забезпечує велику гнучкість в проектуванні загальної архітектури програмного забезпечення
IDE	(англ. Integrated Development Environment) інтегроване середовище розробки
RPC	(англ. Remote Procedure Call, Виклик віддалених процедур) – технологія, що дозволяє програмі на одному комп’ютері запускати функції (процедури) програми на іншому комп’ютері.
FSM	(англ. Finite State Machine) скінченний автомат, машина зі скінченною кількістю станів
PDA	(англ. Pushdown Automaton) автомат з магазинною пам'яттю, скінченний автомат, що додатково використовує стек для зберігання станів
FPS	(англ. frames per second, частота кадрів – швидкість з якою пристрій, що формує зображення відображає послідовні зображення, що мають назву кадри)
SDK	(англ. Software Development Kit) — набір із засобів розробки, утиліт і документації, за допомогою якого програмісти можуть створювати програми за визначеною технологією

WWDC	(англ. Worldwide Developers Conference) щорічна конференція розробників для платформи Macintosh
UI	(англ. User Interface) інтерфейс користувача у графічних системах
ІШІ	ігровий штучний інтелект

ВСТУП

З моменту появи персональних комп'ютерів, та , особливо, з того часу як вони стали доступними для значної частини населення, їх значення та вплив на життя сучасної людини без упину зростало, разом із списком сфер застосування. Ігрова індустрія та інтерактивні відео розваги швидко зайняли місце серед технологій з найбільш стрімким розвитком. За останні роки вони настільки розрослися, що починають поступово витісняти колишніх безсуперечних лідерів у сфері розваг, таких як кіно. Це можна пояснити вкрай широким та багатим арсеналом інструментів у розпорядженні ігрових дизайнерів та розробників, адже гра може являти собою величезну комплексну систему речей, що впливають на чималий спектр відчуттів гравця.

За допомогою графічного ряду можна відобразити абсолютно будь-яку сцену, не обмежуючись фізичними законами, правилами або порядками реального світу, єдині рамки – фантазія дизайнера та вміння художника. Через різні стилі та вміло підібрані кольорові палітри візуального ряду можна передати загальну атмосферу місця, навколишнього світу або події, що постають перед очима гравця. Фонова музика також може мати надзвичайно великий вплив на сприйняття гравця, багато хто вважає що майже 30 відсотків вражень від гри становить один лише саундтрек (наприклад музика композиторів Hans Zimmer та Mick Gordon стала “візитною карткою” ігор CoD Modern Warfare та DOOM).

Ігровий процес (геймплей) – власне один із найголовніших інструментів та характерна риса, яка забезпечує абсолютно унікальний досвід, адже ти власними діями здатен впливати на хід сюжету та стан віртуального світу. Не менш важлива складова, яка присутня в тій чи іншій формі в більшості ігор – сюжет, історія яку автор може розповісти гравцю через взаємодію усіх вище вказаних інструментів, або лише за допомогою одного з них.

Внаслідок цього ігри можуть нести в собі складні ідеї, наводити на обмірковування різних психологічних питань, передавати

найрізноманітніший спектр почуттів та емоцій, через що їх можна сміливо вважати одним із видів сучасного мистецтва. Однак, чимало з них лишаються вірними старим канонам жанру і фокусуються на розважальній частині.

Актуальність теми

Ігрова індустрія на даний момент є вкрай популярною, а різноманітність жанрів, підходів та ідей дозволяє кожному гравцю знайти щось для себе. При цьому ігрове ком'юніті (спільнота) увесь час зростає. Чималу роль у популяризації зіграли ігри в жанрі інтерактивного кіно та прості мобільні аркади, оскільки вони вкрай прості для входження новачків, що вперше доторкнулися до відеоігор.

Мета і задачі дослідження

Об’єктом дипломної роботи є створення відеогри в жанрі Аркада. Вибір зумовлений здебільшого бажанням розібратися у новій та незнайомій для себе сфері, ознайомитися з проблематикою створення відеоігор та штучного інтелекту. В якості основного інструменту розробки буде взято готовий ігровий рушій, який пропонує рішення багатьох базових завдань, таких як високоефективна взаємодія з апаратним забезпеченням користувача, компіляцію проекту, тощо. Також він надає багато зручних інструментів для створення логіки гри на високому рівні, що дозволяє фокусуватися на створенні кінцевого продукту.

Для досягнення поставленої мети необхідно:

- Провести аналіз існуючих ігрових рушіїв, їх переваги та недоліки, їх вбудовані інструменти для розробки. На основі отриманої інформації обрати найбільш оптимальне рішення для реалізації даної гри.

- Перегляд існуючих ігор в даному жанрі, виділення цікавих ідей, що в них реалізовані, а також аналіз їх недоліків, яких варто уникнути
- Обрати візуальний стиль та загальну атмосферу
- Перегляд можливих реалізацій ігрового циклу та задання ігрових правил
- Перегляд існуючих підходів до розробки штучного інтелекту та мережевої гри
- Створити прототип даної гри на обраному рушії, базовий штучний інтелект
- Налагодження зв'язку між декількома гравцями
- Компіляція проекту під декілька основних платформ (Linux, Mac OS, Windows)

РОЗДІЛ 1.
ПРЕДСТАВЛЕННЯ ГОТОВИХ РІШЕНЬ

1.1 Вступ

Загальна ідея ігрового рушія дуже проста. Він є основою будь-якої гри, яка цілком відповідає за її технічну частину. По суті це програмний інструмент (платформа), що бере на себе складні задачі по взаємодії з апаратною частиною кінцевого користувача та надає арсенал інструментів для задоволення більшості потреб розробки в вигляді зручного графічного або програмного інтерфейсу. Крім цього, сучасні ігрові рушії, здебільшого, вміють збирати та оптимізувати проект під найбільш популярні операційні системи, а також для спеціалізованих ігрових приставок (Xbox, Playstation, Switch, тощо) Великі ігрові компанії зазвичай виділяють чимало ресурсів на розробку власного рушія, вузькоспеціалізованого під потреби їх проектів.

Стандартний набір інструментів, що надається популярними ігровими рушіями, зазвичай включає в себе наступне:

1) Графічний рушій – програмне забезпечення, яке в реальному часі малює двовимірну або тривимірну комп’ютерну графіку;
2) Фізичний рушій – програмний рушій, що займається симуляцією законів реального світу (або законів, створених спеціально для гри), та повідомляє про колізії та зіткнення між об’єктами, що задіяні в цій симуляції;
3) Звукова система – забезпечує відтворення та симуляцію поширення звуку у віртуальному просторі;
4) Скриптова мова, що заточена під ефективне програмування ігрової логіки на конкретному ігровому рушії;
5) Інструмент для створення анімацій;
6) Набір інструментів для створення простого штучного інтелекту;

7) Керування пам’яттю та багатопоточністю;
8) Інструменти для мережевого програмування, за допомогою яких можна створювати багатокористувацькі (англ. “multiplayer”) ігри та синхронізувати збережені дані;
9) Граф сцени та текстовий редактор

Однак, існують також ігрові рушії, що заточені під вирішення якоїсь конкретної із вищевказаних задач, наприклад Havok та Box2D – для обробки фізичних явищ, Bink для обробки та створення відеороликів гри, Miles Sound System для обробки звуку, тощо. [1][8]

1.2 Загальні відомості про ігри

Гра — діяльність людини з моделювання іншого роду діяльності, здебільшого має розважальну чи навчальну мету. Вона відрізняється від роботи тим, що не ставить перед собою жодної безпосередньо корисної мети (хоча сама гра може мати власну корисність), а також тісно межує з мистецтвом (В США, наприклад, комп’ютерні ігри офіційно визнані окремим видом мистецтва з 2011), хоч зазвичай і не створює художніх цінностей.

Існує безліч різноманітних видів ігор. Одним із таких і є Аркада – жанр ігор, в якому наявна маніпуляція відносно невеликим набором предметів з простим набором загальних правил. Безпосередню класифікацію доволі складно виділити, оскільки це не лише жанр, а ще й цілий ігровий напрямок. Відеогру точно можна назвати "аркадною", якщо вона напряму портована з автомата або є концептуально подібною до ігор з автоматів. Наприклад, до таких можна віднести всі проекти жанрів "файтинг" (англ. fighting), частину ігор жанру "гонки" (англ. racing) та "шутер" (англ. shooter).

Astroboii – абстрактна аркадна гра, метою якої є накопичення балів за певні дії гравця.

При створенні ігрового рушія необхідно розглянути ігри з точки зору теорії ігор. Теорія ігор – математичний метод вивчення оптимальних стратегій в іграх.

Кооперативні та некооперативні ігри

Кооперативна гра – така гра, в якій гравці можуть об’єднуватись в групи, взявши на себе деякі зобов’язання перед іншими гравцями і координуючи свої дії. В некооперативних іграх кожен грає сам за себе.

Паралельні та послідовні ігри

В паралельних іграх гравці ходять одночасно, або вони не мають безпосередньої інформації про ходи решти гравців, поки всі не зроблять свій хід. В послідовних іграх гравці можуть робити ходи в напередодні визначеному порядку, але при цьому вони отримують деяку інформацію про ходи інших. Ця інформація може бути неповною, наприклад, гравець може дізнатися, що його опонент із десяти стратегій точно не вибрав п'яту, нічого не знаючи про решту [12].

Ігри з повною або неповною інформацією

Гру вважають з повною інформацією, якщо:

- Гравці впливають на ситуацію дискретними діями – ходами, порядок ходів визначений правилами і абсолютно не залежить від таких параметрів, як швидкість реакції гравця (або швидкість апаратної платформи користувача в контексті відеогри).
- В будь-який момент часу всі гравці мають повну інформацію про стан гри, тобто про позиції та всі можливі ходи будь-якого з гравців.

Детерміновані та стохастичні ігри

Якщо всі правила, компоненти та механіки гри повністю незалежні від елементу випадковості – така гра називається детермінованою. Наприклад, шахи є детермінованою грою, а ось нарди або монополія – ні, адже для наступного кроку необхідно кидати гральні кості.

Ігри з нульовою або ненульовою сумою

Ігри, в яких, при переході від однієї ситуації до іншої, збільшення виграшу одного з гравців напряму впливає на зменшення виграшу іншого, називаються іграми з нульовою сумою. Таким чином, сума виграшів в будь-якій ситуації є константною (зазвичай, можна вважати, що вона дорівнює нулю). Прикладом такої гри є покер, де переможець забирає усі фішки опонентів. Для створення, наприклад, ігрового штучного інтелекту для настільних ігор з 2 гравцями, їх вважають іграми з нульовою сумою (якщо перший гравець перемагає, другий – програє, і навпаки).

З наведеного вище можна зробити наступний висновок - Astroboii це некооперативна, паралельна, неперервна, стохастична гра з ненульовою сумою для 1 або більше гравців. [12]

Ігровий процес

Ігровий процес - це специфічний спосіб взаємодії людини з відеогрою. Це шаблон, визначений правилами гри, зв'язок між нею та гравцем, виклики та їх подолання, сюжет та зв'язок гравця з ним. Ігровий процес є однією з основних складових будь-якої гри.

Грабельність

Грабельність (англ. Playability) є загальним показником якості ігрового процесу. Вона визначається як набір властивостей, які описують досвід гравця від взаємодії з певною конкретною ігровою системою, головна мета

якої - забезпечити задоволення та розважити гравця, коли він грає один або в компанії. Грабельність характеризується багатьма різними атрибутами та властивостями для оцінки досвіду гравця:

1) Задоволеність – ступінь прийняття або задоволення від проходження гри або взаємодії з її конкретною частиною, наприклад ігровою механікою, графікою, інтерфейсом користувача, сюжетом, тощо. Це дуже суб'єктивний критерій, через що його вкрай важко оцінити. На результат сильно впливають особисті вподобання та погляди гравця на певні ігрові елементи, такі як характер та зовнішній вигляд головного героя (героїв), візуальний стиль, історія віртуального світу та сюжет, ігровий рівень складності, тощо
2) Крива входження (англ. learning curve) – підхід гри до пояснення ігрових законів та правил. Даний критерій дуже сильно залежить від рішення ігрових дизайнерів та геймплею в цілому. Багато ігор надають чіткі інструкції, що описують правила, за якими працює віртуальний світ, та мають пологу криву входження, де кожен новий ігровий елемент або правило вводиться поетапно та інтуїтивно (іноді для цього відводиться навіть спеціальний сюжетний етап). Однак існує і повністю протилежний підхід, який , наприклад, активно використовується в іграх від студії FromSoftware (Dark Souls, Sekiro). В такому випадку гравця випускають в віртуальний світ самого, без будь-яких прямих пояснень як працюють ігрові системи з боку самої гри. Тоді пошук способів взаємодії (методом проб і помилок, через дослідження та аналіз навколишнього середовища, діалоги, опис предметів, тощо) стає повноцінною ігровою механікою.
3) Ефективність - необхідний час та ресурси, щоб запропонувати гравцям розважальні активності, поки вони займаються виконанням поставлених грою завдань і прямують до остаточної мети (фіналу). Відеогра, що здатна привернути увагу гравця з першої миті та

тримати його зацікавленим до самого кінця, вважається ефективною. Ефективність може бути проаналізована як правильне використання кривої складності протягом гри, правильне структурування цілей та завдань, хороша адаптація вводу користувача до впливу (дії) у грі.

4) Імерсивність (занурення) – здатність повірити у вміст (оточення, події, героїв, тощо) відеогри та інтегрувати гравця у віртуальний ігровий світ. Занурення провокує те, що гравець виглядає втягнутим у віртуальний світ, ніби стаючи його частиною і взаємодіє з ним, оскільки достатньою мірою вірить у віртуальний світ, що показаний у грі, з його законами та правилами, що характеризують цей світ. Відеогра має хороший рівень занурення, коли вона утримує рівновагу між запропонованими викликами та необхідними можливостями гравця для її подолання.
5) Мотивація - характеристики, які провокують гравця до здійснення конкретних дій та зберігаються в них до їх кульмінації. Для отримання високого ступеня мотивації гра повинна мати набір ресурсів для забезпечення наполегливості гравців (утримання цікавості) у виконанні дій для подолання ігрових викликів. Під цим розуміється набір факторів, що забезпечують позитивний відгук при інтерпретації ігрового процесу, зосереджуючи гравця на запропонованих викликах, демонструючи актуальність цілей, які потрібно досягти, і винагороджувати за виклики, сприяючи впевненості гравця в них і задоволенні від їх подолання.
6) Емоційність - мимовільний імпульс, що виникає у відповідь на якийсь стимул з боку відеоігри та викликає певні почуття або певну реакцію. Використання емоцій у відеоіграх допомагає отримати найкращий досвід гравця та приводить до різних емоційних станів: щастя, страху, інтриги, цікавості, смутку, тощо. Це досягається завдяки вмілому використанню ігрових викликів, історії, естетичного

вигляду (візуального стилю) або музичної композиції, які здатні впливати, змушувати посміхатися або плакати гравця. Велика перевага відеоігор полягає в тому, що вони можуть за короткий проміжок часу викликати у гравців різні почуття, деякі з яких важко отримати щодня в реальному світі.

7) Соціалізація - ступінь набору ігрових атрибутів, елементів та ресурсів, що сприяють соціальному фактору ігрового досвіду в групі. Такий досвід провокує оцінку відеоігри по-іншому, завдяки відносинам, встановленим з іншими гравцями або з іншими персонажами гри, які допомагають гравцеві вирішувати ігрові завдання спільно, конкурентно чи кооперативно. Ігрова соціалізація дозволяє гравцям мати абсолютно інший досвід гри, коли вони грають з іншими людьми та сприяє новим соціальним відносинам завдяки взаємодії між ними. На додаток до цього, соціалізація присутня і в тому, як соціальні зв'язки, які ми маємо, проектуються з групою в персонажах відеоігри та контексті, в якому гра реалізується. Наприклад, вибір гравця для підключення або обміну чимось, взаємодія, отримання інформації, прохання про допомогу або проведення переговорів щодо деяких предметів, і як наш вплив з іншим персонажем позитивний чи негативний для досягнення ігрових цілей. Для сприяння соціальному фактору доцільно розробити нові спільні виклики, які допомагають гравцям інтегруватися та задовольнятися новими ігровими правилами та завданнями, створюючи набір колективних емоцій, де гравці (або персонажі) заохочують та мотивують себе подолати колективні виклики.

Для того щоб гра була успішною, необхідно правильно реалізувати в ній хоча б один із аспектів.

1.3 Аркадні та казуальні ігри

Аркадні ігри часто мають короткі рівні, прості та інтуїтивні, веселі схеми управління та швидко наростаючий рівень складності (easy to play, hard to master). Це здебільшого пов’язано з історичного початку Аркадних ігор. Їх предки - ігри для аркадних автоматів, де гравець, по суті, орендує автомат до тих пір, поки його аватар у грі може залишатися живим (або поки у них є в запасі спеціальні жетони). Цим і пояснюється проста головна механіка, та швидко наростаюча складність. Їх сучасне трактування здебільшого доволі сильно відрізняється від оригіналу, зазвичай неявно переходячи у жанр казуальних ігор (хоча перший такий перехід за грою Pac-Man), для якого також характерний набір простих, однак цікавих та затягуючих, основних правил та механік, але крива складності набагато плавніша.

Отже можна виділити наступні характерні риси жанру:

- Веселий, простий геймплей, який легко зрозуміти та освоїти.
- Простий користувальницький інтерфейс, зазвичай адаптований під мобільні пристрої.
- Короткі сеанси, внаслідок чого гру також можна проходити під час перерв на роботі, поїздки в громадському транспорті або під час очікування в черзі тощо
- Простий візуальний стиль
- Часто є спрощеним варіантом гри іншого жанру, або містить подібні ігрові механіки (через що може містити чимало елементів, характерних для інших жанрів)

1.4 Відомі існуючі ігри у жанрі Аркада

Space Invaders

Space Invaders – комп'ютерна гра, розроблена Tomohiro Nishikado в 1978 році. Вона заснувала новий ігровий жанр «перестріляти їх усіх» (англ. Shoot 'em up), який є одним із піджанрів Аркади. Основна механіка цієї гри полягала у протистоянні одного гравця і великої кількості ворогів, що постійно спускалися з верхнього краю екрану. З кожним наступним рівнем їх швидкість руху, темп стрільби та влучність збільшувались. Не зважаючи на свою просто задумку, гра мала вкрай великий комерційний успіх. Настільки великий для свого часу, що навіть викликав нестачу монет для ігрових автоматів в Японії.

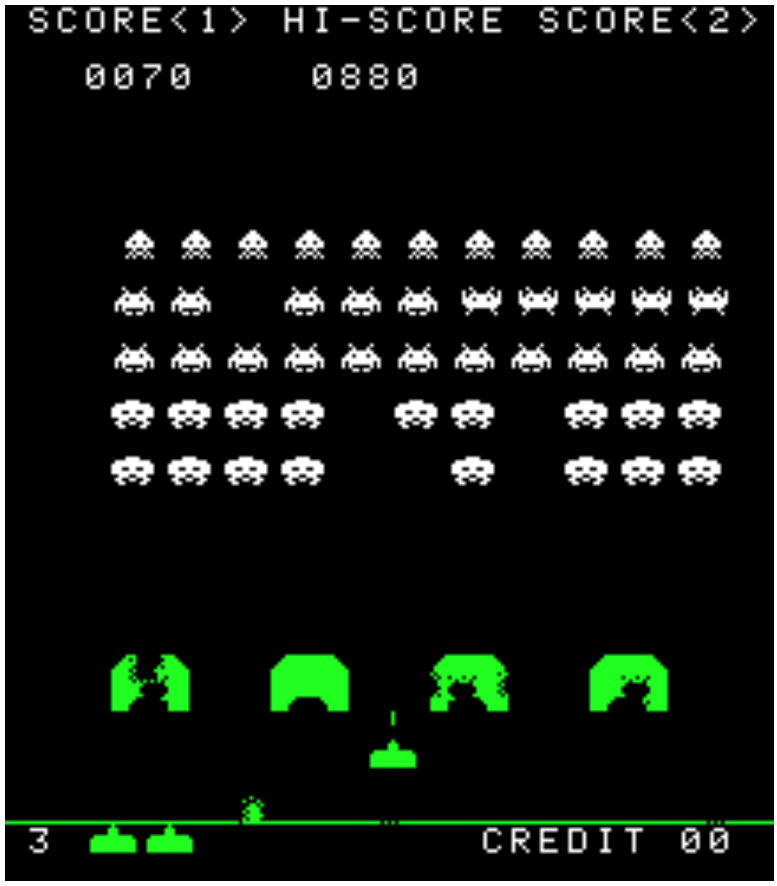


Рисунок 1.1 – Space Invaders

Asteroids

Ateroids – це космічна аркадна стрілялка (англ. shooter), розроблена Лайлом Рейнсом, Едом Логом та Домініком Уолшем і випущена в 1979 році компанією Atari. Гравець має під контролем один космічний корабель посеред поля астероїдів, в якому час від часу пролітають вороже налаштовані літаючі тарілки. Головна мета гри – стріляти і знищувати астероїди та ворожі кораблі, уникаючи зіткнень з ними та ухиляючись від ворожих пострілів. З плином часу, протягом однієї ігрової сесії, швидкість та кількість астероїдів невпинно зростає. Дана гра отримала широке розповсюдження та значний комерційний успіх, було продано більше 70,000 копій. В 1980 вона була портована для домашніх систем Atari і продана в кількості більше трьох мільйонів.

Рисунок 1.2 – Asteroids

ВИСНОВОК ДО РОЗДІЛУ 1

Інформація отримана в цьому розділі вплинула на остаточні рішення, пов’язані з набором технологій потрібних для реалізації, дистрибуції та підтримки гри Astroboii. Дослідження області теорії ігор дозволило формально описати гру Astroboii:

Astroboii – некооперативна, паралельна, неперервна, стохастична гра з ненульовою сумою для 1 або більше гравців. За основну ідею взята задумка Asteroids та Space Invaders.

Отже, спираючись на досвід та ідеї попередніх реалізацій аркадних ігор та аналіз ігрових принципів (критеріїв грабельності) перед грою було поставлено наступні задачі:

1) Прості в освоєнні та веселі базові ігрові механіки.

2) Простий візуальний стиль, що не напружує.

3) можливість грати із штучним інтелектом.

4) Простий штучний інтелект для ворогів.

5) Встановлення переможця та переможеного.

6) Графічний інтерфейс для відображення стану гри.

7) Налагодження зв’язку між двома реальними гравцями (мережева складова).

РОЗДІЛ 2
АНАЛІЗ ЗАСОБІВ ДЛЯ РОЗРОБКИ КОМП'ЮТЕРНОЇ ГРИ

2.1 Аналіз сучасних реалізацій ігрових рушіїв

В даному розділі будуть розглянуті популярні доступні ігрові рушії, проаналізовані переваги та недоліки їх використання для створення Astroboii зокрема, та комп'ютерних ігор загалом.

2.1.1 Unity

Unity – це кросплатформний ігровий рушій, розроблений американською компанією Unity Technologies, вперше представлений у червні 2005 року на WWDC як рішення для створення ігор для платформи Mac OS X (на даний момент може збирати проект для роботи на більш ніж 25 різних платформах). Даний інструмент також є повноцінним середовищем для розробки відеоігор, адже має графічний інтерфейс, вбудований текстовий редактор та відладжувач (дебагер), прості інструменти для роботи з текстурами та анімаціями [4].

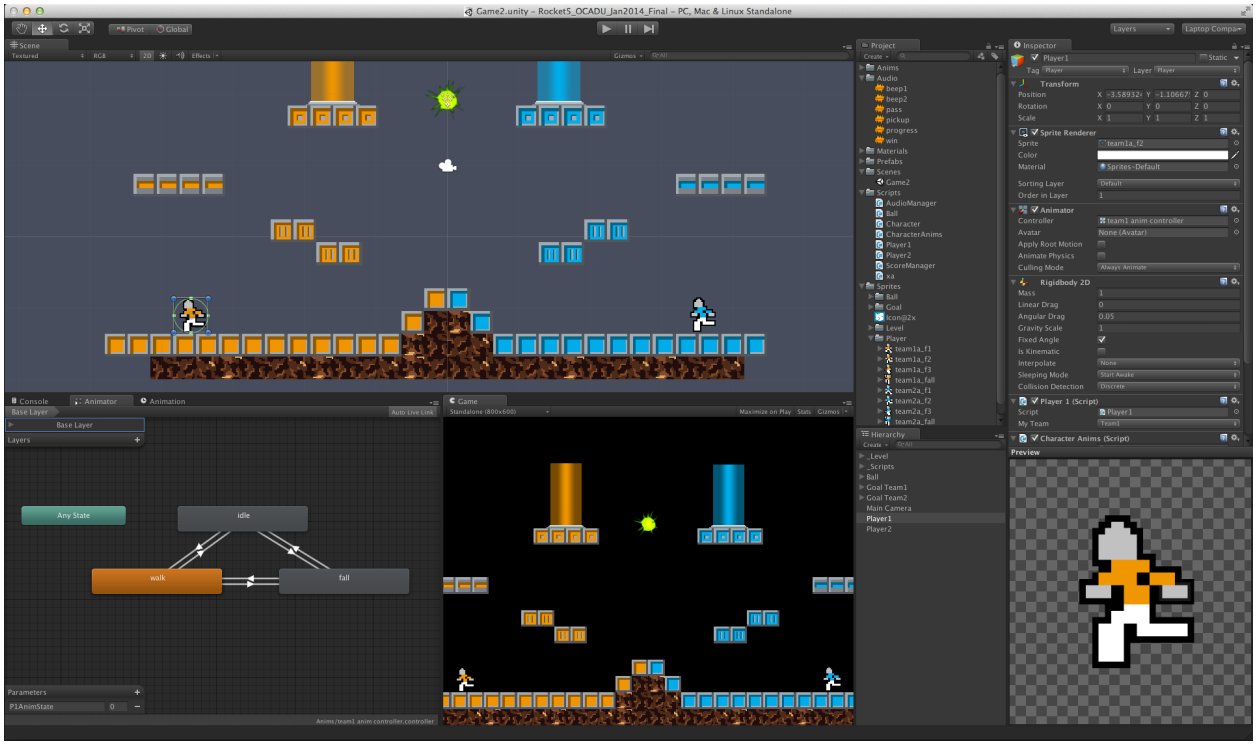


Рисунок 2.1 – Інтерфейс редактора Unity

До плюсів даного варіанту можна віднести наступне:

1) Доволі зручний графічний інтерфейс редактора з підтримкою Drag & Drop функціоналу, та який можна гнучко налаштувати під власні потреби
2) Для написання скриптів використовується C#, що звільняє від потреби вивчати нову мову виключно для взаємодії з даним рушієм (якщо розробник уже знає C#), а також дозволяє використовувати в розробці чималий арсенал існуючих і добре протестованих інструментів для даної мови
3) Для систематизації архітектури використовується ECS, що забезпечує велику гнучкість в проектуванні гри
4) Велика кількість навчальних матеріалів про створення ігор за допомогою цього рушія
5) Велика доступна бібліотека заготовок (безкоштовні або платні набори текстур та моделей) та плагінів (допоміжна стороння бібліотека для розширення базового функціоналу рушія), що дозволяє сильно полегшити процес розробки
6) Можливість збірки проекту під усі популярні платформи

Серед мінусів можна відмітити:

1) Доволі висока вимогливість до апаратних потужностей користувача, навіть для простих проектів. Без необхідної оптимізації гра може вийти повільною та ресурсномісткою
2) Графічний рушій в Unity не підтримує справжнє 2D, існуюча реалізація використовує 3D простір з виставленням Z-осі в 1, що додає багато накладних витрат апаратних ресурсів
3) Ігри створені на Unity займають багато дискового простору. Навіть просто гра може мати розмір в декілька сотень мегабайт.

4) Відсутність вбудованих інструментів для створення штучного інтелекту
5) Складність в систематизації великих сцен

З наведеного вище можна зробити висновок, що, хоч Unity має чималий набір зручних інструментів, він занадто важкий для реалізації Astroboii.

Ігри розроблені на Unity [4]:

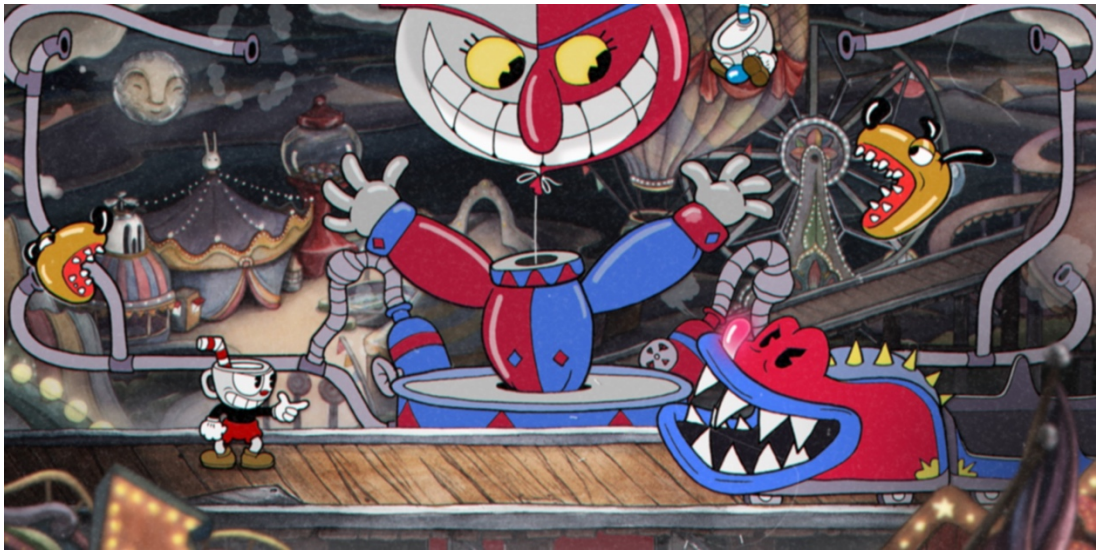


Рисунок 2.2 – Cuphead

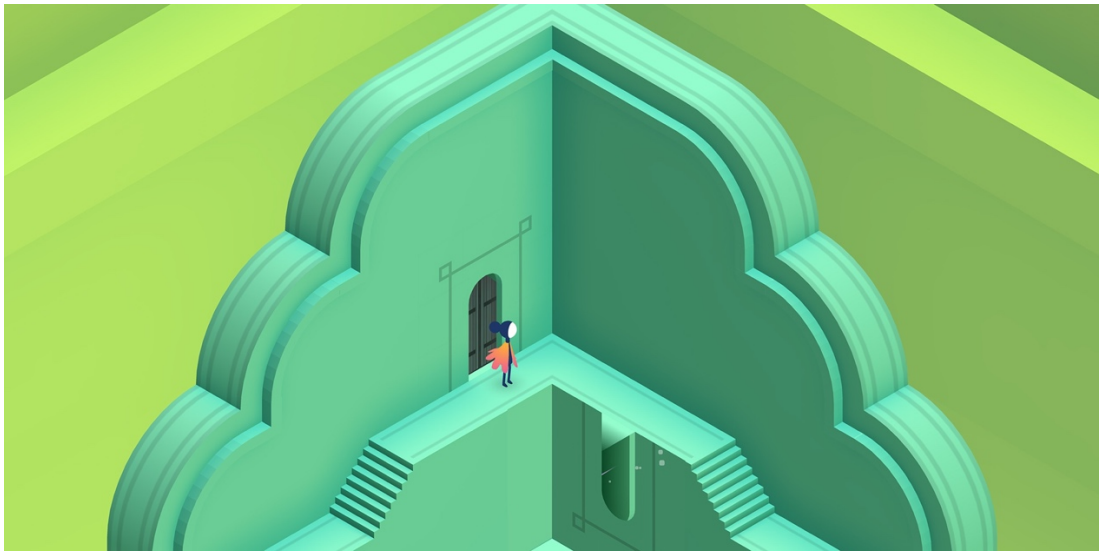


Рисунок 2.3 – Monument Valley 2

Рисунок 2.4 – Ori and The Blind Forest

2.1.2 Godot

Godot – відкритий багатоплатформовий ігровий рушій під ліцензією MIT. Даний інструмент розроблюється співавторством Godot Engine Community та має повність відкритий сирцевий код. Як і Unity є повноцінним середовищем для розробки ігор. Крім цього має вбудований функціонал для створення простих текстур, матеріалів та шейдерів. Також характерною рисою є вбудований створювач анімацій, який дозволяє анімувати будь-який компонент гри. Задача, яку поставили перед собою розробники Godot, створити максимально інтегроване та самодостатнє середовище для розробки ігор, що цілком пояснює такий багатий арсенал вбудованих інструментів. [8]

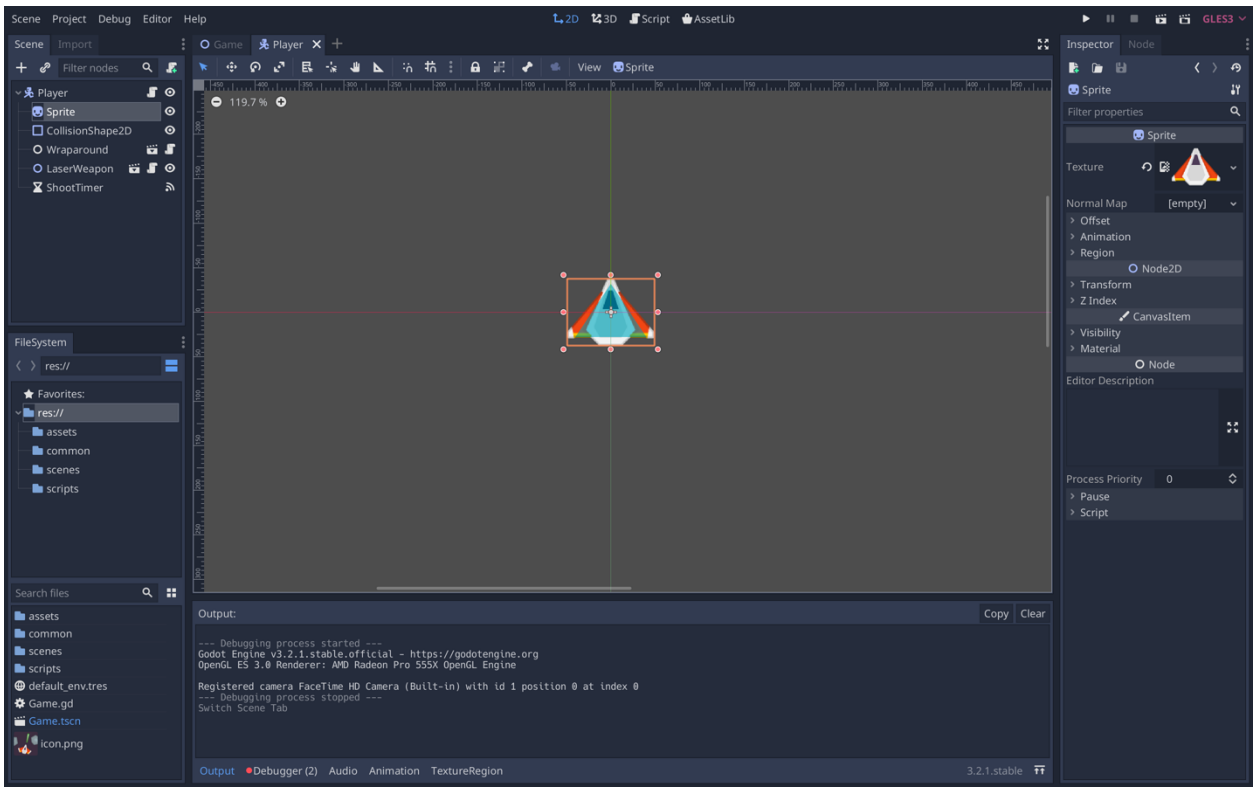


Рисунок 2.5 – Інтерфейс редактора Godot

До плюсів даного варіанту можна віднести наступне:

1) Дуже зручний графічний інтерфейс редактора з підтримкою Drag & Drop функціоналу, та який можна гнучко налаштувати під власні потреби. Крім того, на відміну від Unity, в базовий функціонал входить можливість змінити тему усього інтерфейсу, або розробити свою для максимально комфортної роботи (в Unity така можливість є лише у платній версії)
2) Для написання ігрової логіки можна використовувати C++, C# та власну скриптову мову gdscript, дуже схожу на python. Крім цього рушій надає інтерфейс, через який з ним може взаємодіяти будь-яка стороння мова програмування, однак, у таком випадку треба буде дописати свої конфігурації для збірки проекту. Така гнучкість дозволяє обрати програмісту найзручніший для себе інструмент і працювати з ним.

3) Редактор підтримує візуальну мову програмування, що може бути дуже зручним інструментом для швидкої прототипізації.
4) Загальна архітектура рушія побудована навколо концепції дерева з наслідуваних "сцен". Кожен компонент сцени (ноду) можна в будь-який момент зробити повноцінною сценою. Така гнучкість дозволяє легко вносити зміни будь-якого масштабу в архітектуру проекту, розширювати її елементи в будь-який бік та працювати із складними сценами на рівні простих абстракцій. На практиці, для розробки та структуризації проекту, такий підхід виявився значно зручнішим за ECS.
5) Потужний та простий у використанні інструмент для створення інтерфейсу користувача (англ. UI)
6) Графічний рушій підтримує справжнє 2D, а не емулює його в 3D.
7) Вбудований інструмент для створення анімацій, а також дерево анімацій, що дозволяє легко задавати складну логіку переходів та програвання анімацій.
8) Інструменти для створення шейдерів та простих текстур.
9) Зібрані проекти, як і сам рушій, мають дуже скромні вимоги до апаратного забезпечення кінцевого користувача та займають небагато дискового простору.
10) Рушій легко інтегрувати з системами контролю версій та сторонніми текстовими редакторами та IDE (VS Code, Rider та інші).

Серед мінусів слід зазначити:

1) Доволі скромна, у порівнянні з Unity, кількість прикладів та навчальних матеріалів по роботі з даним рушієм.
2) Набагато менший, у порівнянні з тим же Unity, набір плагінів та заготовок.

3) Загальна архітектура та систематизація проекту повністю залежить від розробника (що і перевага і недолік одночасно, однак у великому проекті може створити проблеми)
4) Відсутній вбудований модуль для створення штучного інтелекту

Ігри розроблені на Godot [7][8]:

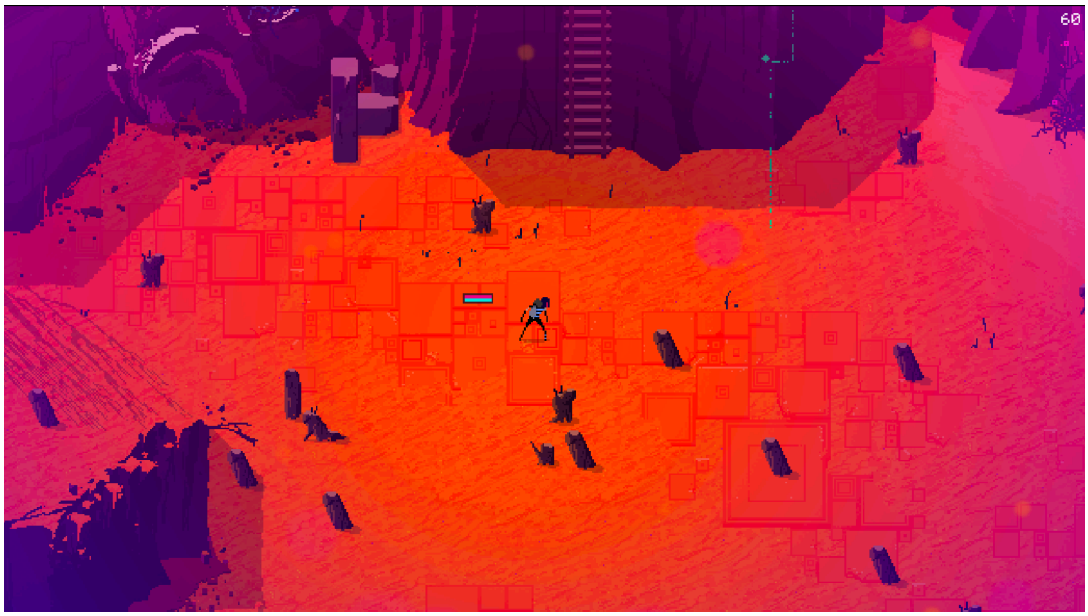


Рисунок 2.6 – Resolutiion

Рисунок 2.7 – Space Ace

Рисунок 2.8 – ProtoCorgi

2.1.3 LÖVE

LÖVE (також відомий як Love2D) — вільно розповсюджуваний кросплатформений фреймворк, призначений для розробки комп'ютерних ігор на мові Lua. Поширюється по ліцензії zlib, яка передбачає вільне використання як у відкритих, так і в комерційних проектах із закритим сирцевим кодом.

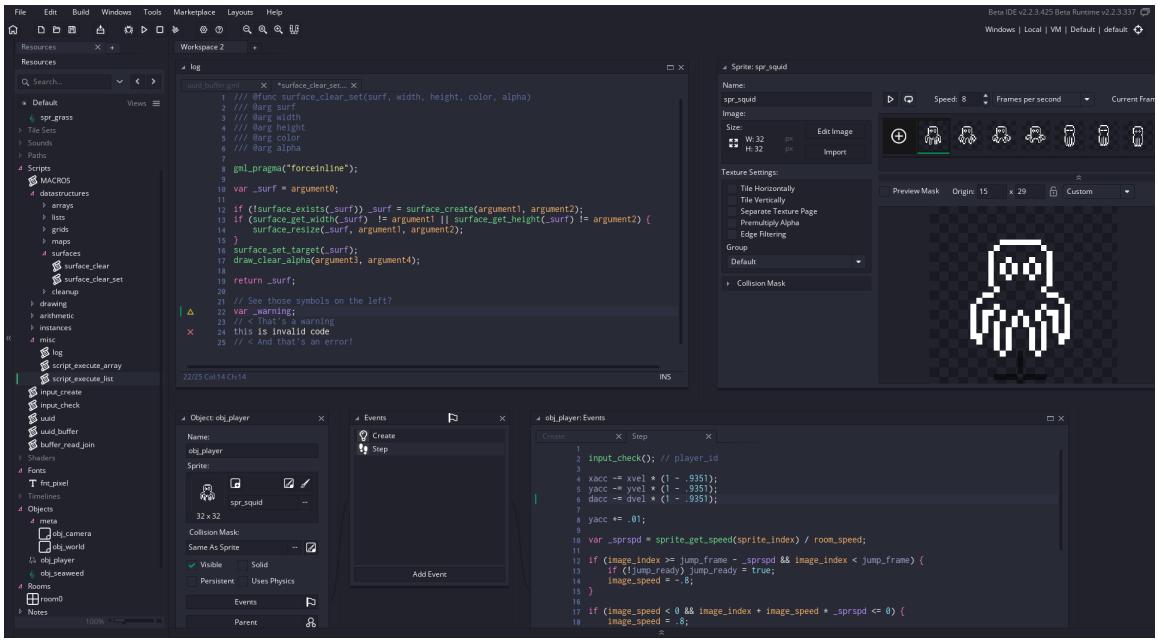


Рисунок 2.9 – Сторонній інтерфейс для взаємодії LÖVE

До переваг даного інструменту можна віднести:

1) Душе швидка прототипізація простих 2D проектів.
2) Зручний інтерфейс для взаємодії з фреймворком.
3) Дуже невибагливий до апаратного забезпечення та займає дуже мало дискового простору.

Серед мінусів слід зазначити:

1) Відсутність графічного інтерфейсу.
2) Відсутність фізичного рушія.
3) Мала кількість навчального матеріалу.
4) Відсутні заготовки, кількість плагінів вкрай невелика.
5) Відсутність модуля для програмування штучного інтелекту.

Ігри розроблені на LÖVE:

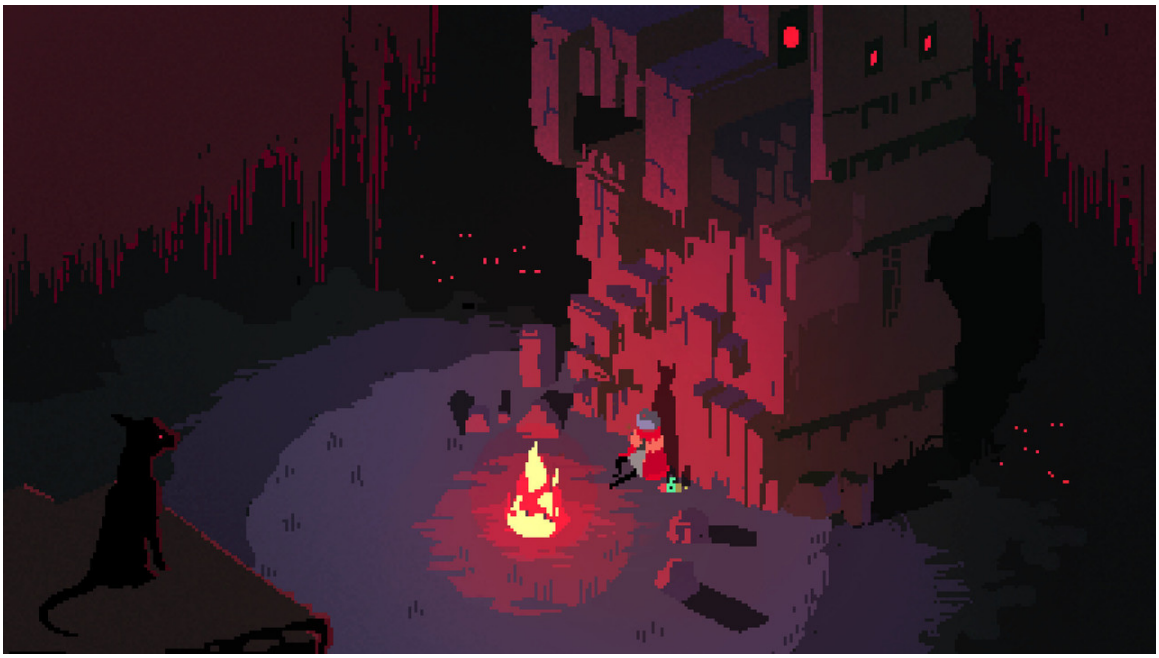


Рисунок 2.10 – Hyper Light Drifter

Рисунок 2.11 – Sea Salt

Рисунок 2.12 – Jet Lancer

2.2 Штучний інтелект у контексті ігрового рушія

Штучний інтелект в основному використовується для відтворення адаптивної або розумної поведінки неігрових персонажів, подібної до людської. Він був невід'ємною частиною процесу розробки ігор з часу їх появи. Його роль поступово розширилася з плином часу та розвитком технології. Так, у сучасних ігрових рушіях штучний інтелект використовується для пошуку найкоротшого шляху, симуляції фізичних явищ природи, моделювання емоційної та фізичної поведінки персонажів. Також він використовується у деяких механіках гри, які заздалегідь не відомі гравцю — збір інформації про його поведінку, процедурна генерація рівнів у процесі проходження гри, тощо.

Термін "Ігровий штучний інтелект" використовується для опису вкрай широкого кола алгоритмів з зовсім різних областей науки: комп'ютерної графіки, інформатики, теорії керування, робототехніки, тощо. Тому він зазвичай не відповідає ознакам "реального штучного інтелекту", таким як машинне навчання, і здебільшого займається автоматизацією певних ігрових процесів або вибором одного із заздалегідь відомих варіантів.

Розробка програмних рушіїв була однією з областей де ІШІ застосовувався з самого початку свого розвитку. Так, ІШІ гри Pong, яка була написана у 1972, часто вигравав супротивників високого рівня. У 1951 році, використовуючи комп'ютер Ferranti Mark 1, Крістофер Старчі створив перший ігровий рушій для гри в шашки, а Дітріх Прінз для гри у шахи. Кульмінацією ігрових рушіїв для настільних ігор буде поразка Гарі Каспарова у грі проти штучного інтелекту Deep Blue у 1997, яка покладе початок ери перемог комп'ютера над людиною.

2.2.1 State Machine (FSM)

FSM – найпростіший член сімейства структур даних теорії автоматів (яке також включає машину Тюрінга). Вона все ще активно використовується для написання простого штучного інтелекту та опису стану ігрового об’єкту (також використовують як основу простого програвача анімацій).

Ідея полягає в наступному:

- Є фіксований набір станів, в яких може знаходитися машина. На наведеному нижче прикладі це стояння, стрибки, полювання та пірнання.
- Машина може одночасно знаходитись лише в одному стані. Наприклад герой не може стрибати і стояти одночасно.
- Послідовність входів або подій надсилається на апарат. У нашому прикладі ця натиск та відпускання кнопки.
- Кожен стан має набір переходів, кожен пов'язаний із вводом та вказує на стан. Коли надходить ввід, якщо він відповідає переходу для поточного стану, машина переходить у стан, на який вказує перехід.

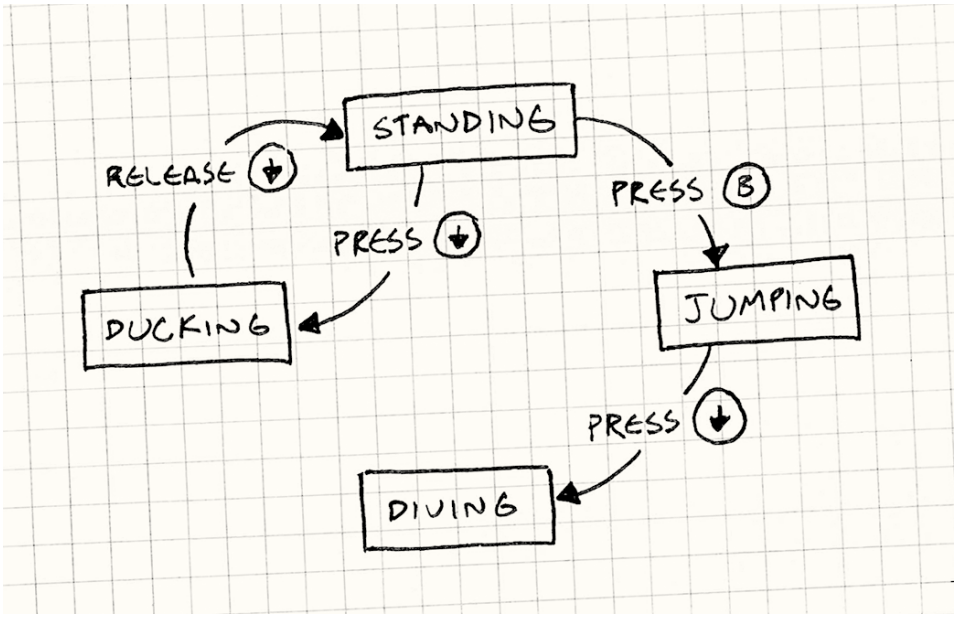


Рисунок 2.12 – Приклад простої FSM

Найбільша перевага FSM – одночасно і його найбільший недолік, простота. Все, що у нас є, - це фіксований набір станів, єдиний поточний стан та деякі фіксовані переходи. Цього швидко стане недостатньо для спроби створити хоч трохи складніший штучний інтелект [15].

2.2.2 Автомат з магазинною пам'яттю (англ. Pushdown automaton)

Існує ще одне поширене розширення для скінченного автомату, яке додатково використовує стек станів. Проблема полягає в тому, що FSM не мають історії. Ви знаєте, в якому стані перебуваєте, але не пам’ятаєте, в якому стані знаходилися. Немає простого способу повернутися до попереднього стану.

В той час, як FSM має єдиний вказівник на стан, автомат з магазинною пам’яттю має стек вказівників. У скінченному автоматі перехід до нового стану замінює попередній. Автомат з магазинною пам’яттю теж має таку можливість, однак він надає ще дві додаткові опції:

- Є можливість покласти новий стан на стек. "Поточний" стан - це завжди той, що знаходиться на вершині стеку. Коли приходить новий стан, він "штовхає" поточний стан далі в стек і записується на вершину, а не просто переписує його.
- Є можливість відкинути верхній стан стеку, тоді його місце займає стан, що раніше був під ним.

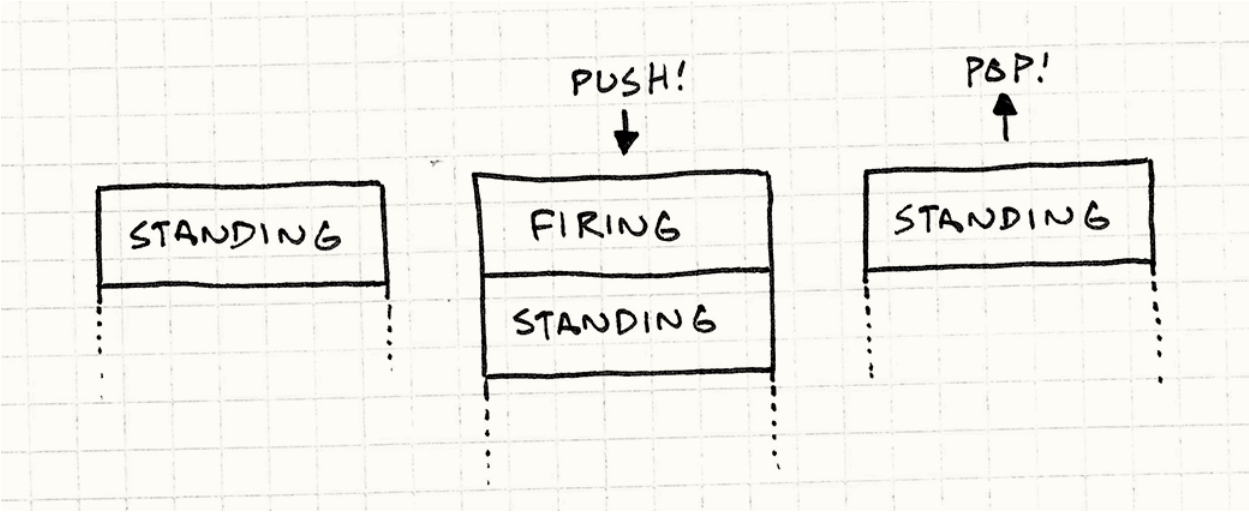


Рисунок 2.13 – Приклад простої PDA

Навіть із таким розширенням, FSM все ще досить обмежені. Однак, це зовсім не означає, що скінченний автомат, автомат з магазинною пам’яттю або інші прості системи безкорисні. Вони є хорошим інструментом моделювання для вирішення певних проблем. Скінченні автомати корисні, коли:

- Є сутність, поведінка якої змінюється на основі певного внутрішнього стану.
- Цей стан можна жорстко виділити в один, із порівняно невеликої кількості різних варіантів.
- Сутність реагує на серію вводів або подій з плином часу.

В іграх вони знаходять чимало застосування, окрім простого штучного інтелекту, наприклад в реалізаціях обробки вводу користувача, навігаційних екранів та меню, парсингу тексту, мережевих протоколів та інших асинхронних поведінок [15].

2.2.3 Дерево поведінки (англ. Behavior Tree)

Головна ідея описується самою назвою. На відміну від скінченних автоматів або інших простих систем, що використовуються для програмування штучного інтелекту, дерево поведінки – це дерево ієрархічних вузлів, які керують процесом прийняття рішень об’єктом під контролем штучного інтелекту. В межах дерева, листки це дійсні команди, що керують підконтрольним об’єктом, а формуючі гілки – різні види службових вузлів, які направляють штучний інтелект по дереву для досягнення послідовності команд, що найкраще підходять для навколишньої ситуації. Дерева можуть бути дуже глибокими, з вузлами, що переходять в піддерева, які виконують певні функції, що дозволяє розробнику створювати окремі бібліотеки поведінок, які, в свою чергу можна комбінувати і поєднувати в ланцюг, для створення дуже складного та правдоподібного штучного інтелекту. Розробка дуже ітеративна, можна почати з формування базової поведінки, а потім створювати і додавати нові гілки, які будуть забезпечувати альтернативні методи досягнення цілей. Крім цього, гілки можуть мати різний ваговий коефіцієнт бажаності (правило визначається розробником), що забезпечує наявність запасної тактики для штучного інтелекту в разі збою або некоректної роботи певної поведінки. Це найсильніша сторона дерева поведінок [2][15].

Ключовим аспектом дерева поведінки є те, що конкретний вузол може займати багато тактів гри для виконання. В базовій реалізації система буде проходити вниз від кореня дерева кожен окремий кадр, перевіряючи кожен вузол вниз по дереву, щоб дізнатися який із них наразі активний, перевіряючи увесь шлях, поки не досягне власне активного вузла. Це дуже неефективний спосіб, особливо коли дерево поведінки стає глибшим та розвивається. Тому реалізація має підтримувати функціонал збереження активних вузлів, для швидшого орієнтування системи (це знімає необхідність повного проходу на кожному такті).

Дерево поведінки складається з декількох типів вузлів, однак деякі базові функції є загальними для будь-якого з них. Одна з таких – повернення одного з трьох статусів (в залежності від реалізації їх може бути більше) [2]:

1) Success
2) Failure
3) Running

Перші два інформують батьків про успішність або провал своєї операції. Третій означає що успіх або невдача ще не визначені, а сам вузол все ще в процесі роботи. Вузол може буди повторно обраний після проходження дерева. Ця функціональність є одною з ключових для дерев поведінки, оскільки дозволяє обробці вузла зберігатися для багатьох тактів гри. Наприклад, вузол “Йти” запропонує статус “Бігти” протягом часу, коли він намагається обчислити ефективний шлях та час, необхідний персонажу для переходу до вказаного місця. Якщо обрахунок маршруту не вдався з будь-якої причини, або виникло якесь інше ускладнення під час ходьби, вузол повертає статус Failure батьківському вузлу, щоб зупинити персонажа. Якщо в будь-який момент поточне місцезнаходження персонажа дорівнює цільовому положенню, то вузол повертає Success, вказуючи на успішно виконану команду “Йти” [13][15].

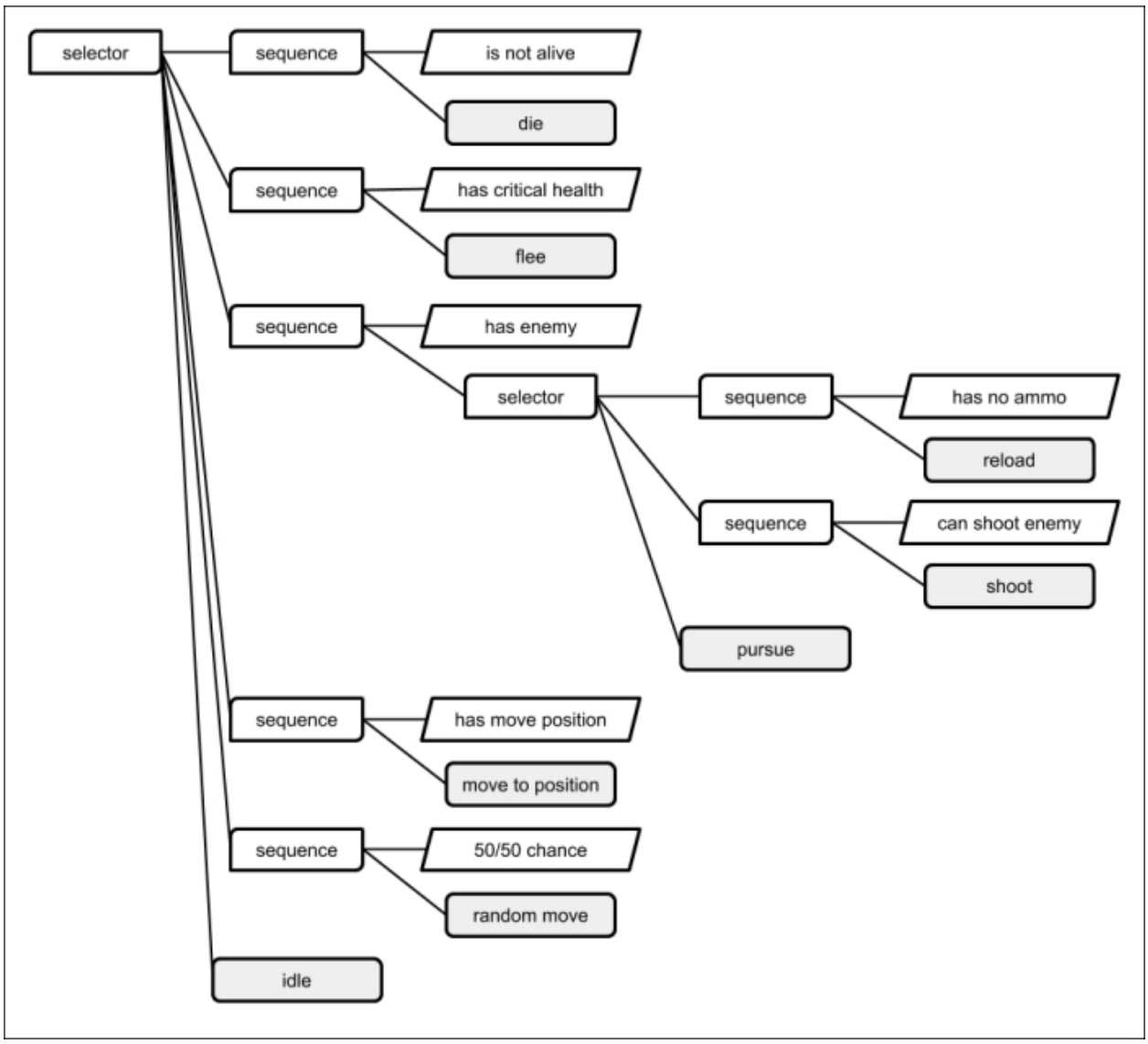


Рисунок 2.14 – Приклад дерева поведінки

ВИСНОВОК ДО РОЗДІЛУ 2

Аналіз існуючих доступних ігрових рушіїв показав, що для розробки Astroboii доцільно використовувати Godot, оскільки він виявився вкрай зручним у використанні і дозволить швидко розробити прототип та саму гру. Також проект на його основі буде невибагливим до апаратного забезпечення користувача і його можна буде легко зібрати під будь-яку поширену платформу.

В якості скриптової мови програмування був обраний gdscript, оскільки вона має найвищий рівень інтеграції з обраним рушієм. Критичні для швидкодії ділянки коду можна винести в окремі модулі GDNative, реалізовані на C++ або Golang.

За основу для реалізації штучного інтелекту буде взятий автомат з магазинною пам’яттю.

РОЗДІЛ 3
РОЗРОБКА ПРОГРАМНОГО ПРОДУКТУ

3.1 Проектування архітектури

Огляд правил гри. Основні правила дуже прості – гравець на своєму зорельоті потрапляє в пояс астероїдів, що обмежений силовим полем. З плином часу поле починає зменшуватись у діаметрі, а кількість ворогів та астероїдів починає невпинно зростати. Єдиний спосіб вибратись та здобути перемогу – зібрати необхідну кількість секретних ключів та ресурсів для здійснення просторового стрибка. Для отримання ресурсів необхідно знищувати певні види астероїдів та видобувати з них корисні матеріали. Секретні ключі генеруються в випадкових локаціях, транспортуються деякими ворожими зорельотами або зберігаються на станціях.

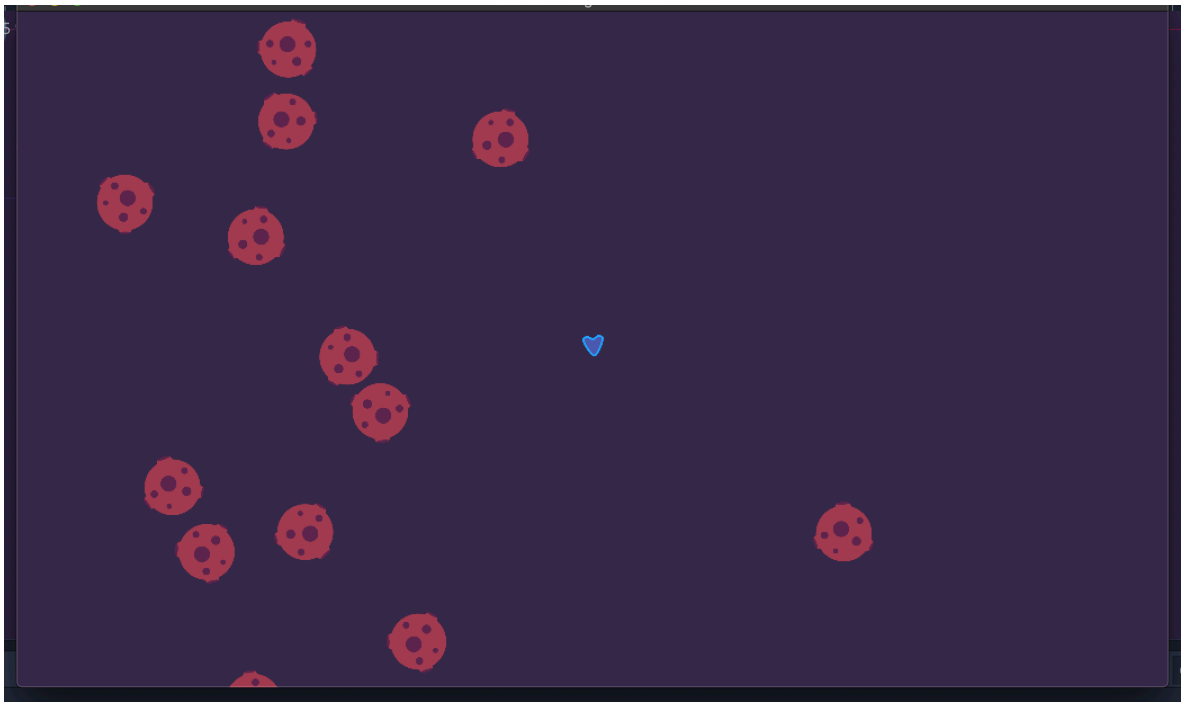


Рисунок 3.1 – Представлення середовища гри

Ігрове середовище міститиме в собі наступні елементи:

- Основне ігрове поле (пояс астероїдів обмежений силовим бар’єром)
- Зорельот під контролем гравця
- Астероїди різних типів
- Різні види бонусів та зброї
- Ворожі зорельоти
- Дослідницькі станції
- Секретні ключі та додаткові ресурси

3.2 Основні рішення для реалізації гри та її компонентів

Основний базовий будівельний блок в Godot – вузол (англ. Node). Він має набір атрибутів та методів, що представлені у вигляді публічного інтерфейсу для взаємодії за допомогою графічного інтерфейсу або коду програми. До кожного вузла можна приєднати ‘скрипт’. Скрипт в даному контексті – програмний код що описує поведінку цього конкретного вузла, може впливати на його атрибути та життєвий цикл. Кожен вузол може бути призначений дочірнім або батьківським для іншого. Таким чином загальна архітектура має вигляд дерева. Батьківський вузол може мати будь-яку кількість дочірніх елементів із єдиною вимогою, щоб усі вони мали унікальне ім’я в межах свого ярусу. Дерево вузлів називається сценою. Сцени можна зберігати як окремий об’єкт та компонувати їх між собою за тими ж правилами, що використовуються для вузлів. Це забезпечує дуже високу гнучкість в архітектурі та моделі даних проектів Godot.

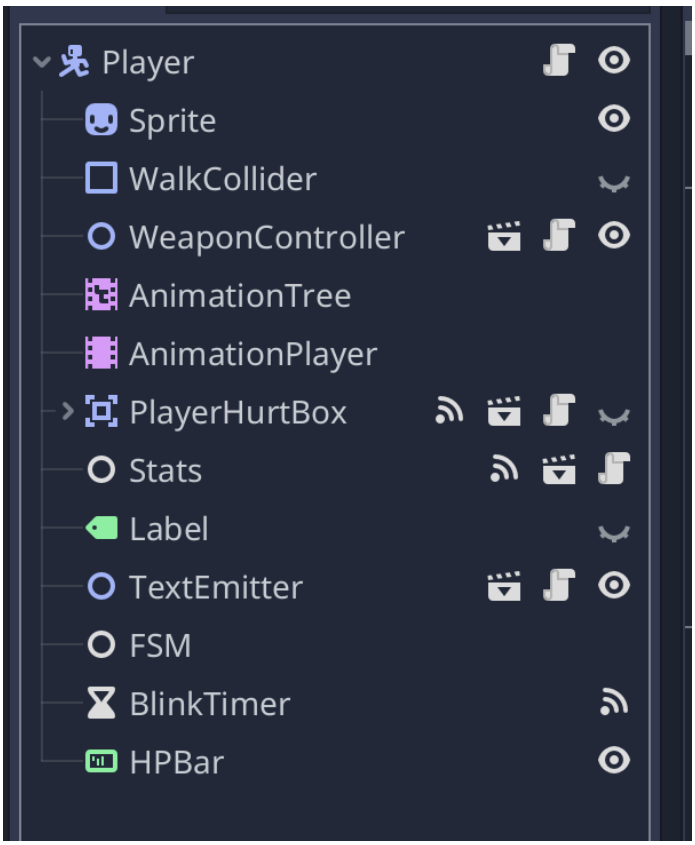


Рисунок 3.2 – Приклад дерева сцен та вузлів (сцена аватару гравця)

Після того, як всі необхідні сцени будуть завантажені в гру, вони отримують повідомлення NOTIFICATION_READY, що викликає відповідні методи _ready для кожного дочірнього елементу. Цей метод є у кожного вузла та містить у собі логіку для створення початкових даних. Для груп вузлів метод _ready викликається у зворотному порядку, починаючи з дочірніх та переходячи до батьківських.

Для опису основної логіки роботи вузла, що залежить від часу (тобто не є статичною) використовуються два методи – _process та _physics_process. Вони як і _ready є стандартними і присутні в кожній вершині. Метод process викликатиметься на кожному кадрі і виконуватиме прописану задачу. Для обробки фізичних задач рекомендується використовувати _physics_process, оскільки його обробка відбувається фіксовану кількість разів в секунду (за замовчуванням 60) і жодним чином не залежить від частоти зміни кадрів.

Вузли також можуть обробляти події введення. При наявності, функція _input буде викликана для кожного вводу, який отримує програма. У багатьох випадках це може вести до надмірного використання ресурсів (якщо не застосовується для простих проектів). В більшості випадків, для вузлів, що не знаходяться під безпосереднім контролем гравця, краще використовувати _unhandled_input. Він викликається лише у тому випадку, коли подію вводу не обробляв ніхто інший. Такий підхід гарантує, що вузол отримує лише події, призначені безпосередньо для нього.

Щоб відслідковувати ієрархію сцен (особливо при наслідуванні сцен в інші сцени), для вузла з присутнім параметром owner встановлюється посилання на батьківський вузол.

Для видалення вузла с гри використовується метод Object.free або queue_free. При цьому всі дочірні вершини також будуть видалені.

Взаємодія вузлів з мережею - після підключення до сервера можна використовувати вбудовану систему RPC (віддалений виклик процедури) для передачі даних по мережі. Звертаючись до RPC по імені методу, він буде викликатися локально, на сервері, який приймає з'єднання, та у всіх підключених клієнтах [3].

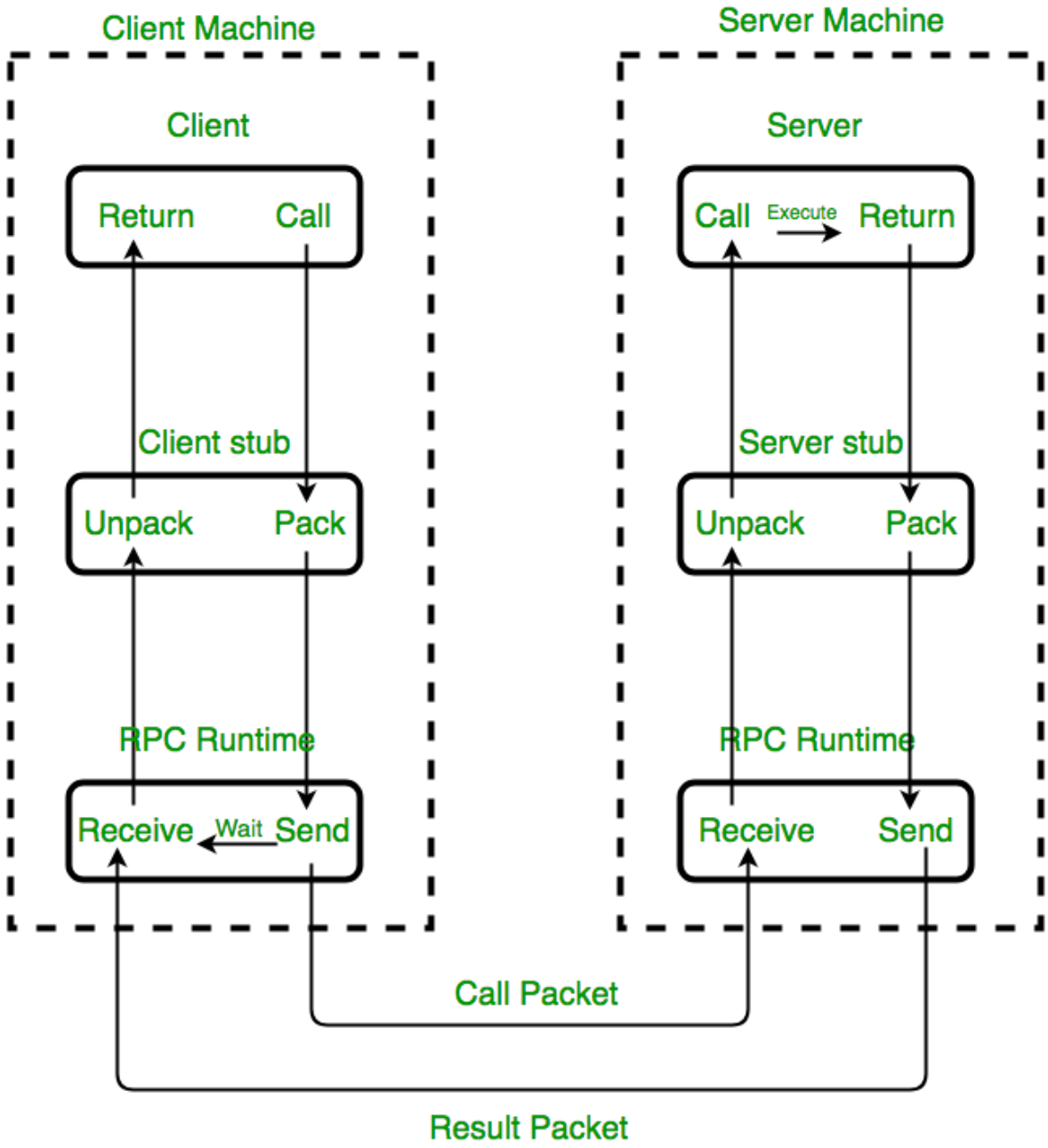


Рисунок 3.3 – Принцип роботи RPC

3.2 Основні інструменти та вузли, що використовуються при розробці проекту

Node

Основний вузол, з якого наслідуються усі інші. Містить лише стандартні методи та параметри [7]. У проекті використовуються наступні функції:

- _enter_tree () virtual – містить у собі логіку, що виконується при входженні (додаванні) даного вузла у сцену.
- _exit_tree () virtual - містить у собі логіку, що виконується при видаленні даного вузла зі сцени.
- _input (InputEvent event) virtual – функція, що буде викликана при надходженні події вводу.
- _unhandled_input (InputEvent event) - функція, що буде викликана при надходженні необробленої події вводу.
- _ready () – відповідає за ініціалізацію початкових даних.
- _process (float delta) – функція, що викликатиметься після кожної зміни кадру
- _physics_process (float delta) – функція, що викликатиметься чітко встановлену кількість разів за секунду
- add_child (Node node) – додавання дочірнього вузла
- get_child (int idx) – отримання даних дочірнього вузла
- get_children () – отримання всіх дочірніх вузлів
- get_tree () – отримати головну батьківську сцену
- queue_free () – видалення вузла с гри

До даного вузла також можна прикріпити скрипт. За допомогою цих найпростіших вершин можна зробити візуальну структуризацію коду. Наприклад, кожен окремий стан скінченного автомата може бути представлений у вигляді вузла Node, а прикріплений скрипт відповідатиме за його безпосередню поведінку.

Node2D

Вузол, який успадковується усіма 2D вершинами, включаючи фізичні об’єкти та спрайти. Має наступні ключові параметри – position, rotation, scale, Z index, що дозволяє переміщувати, збільшувати та обертати даний вузол та всі його дочірні елементи. За допомогою Z index можна вказувати порядок

відображення у графічному рушії [7].

Основі методи, що використовуються у проекті:

- apply_scale (Vector2 ratio) – множить теперішній розмір вузла на заданий вектор
- get_angle_to (Vector2 point) – функція, за допомогою якої можна обрахувати кут в радіанах між даним вузлом та вказаною точкою у двовимірному просторі
- rotate (float radians) – повернути вузол відносно свого теперішнього положення на вказаний кут
- look_at (Vector2 point) – повернути вузол в напрямку вказаної точки

Даний вузол використовується для усіх ігрових компонентів, що мають позицію у двовимірному просторі та не потребують складної логіки, або в якості батьківського елемента для сцени.

Position2D

Такий же вузол як і Node2D, однак має додаткові опції для відладки та візуально відображається у графічному редакторі ігрового рушія.

Sprite

Вузол, що використовується для відображення двовимірної текстури. Може використовувати атлас текстур або кадр з шаблону анімацій. Надає базовий набір можливостей для взаємодії з зображенням за допомогою параметрів centered, flip_h, flip_v, hframes, vframes.

Raycast2D

Даний вузол являє собою лінію від свого початкового положення до точки у двовимірному просторі, що вказана параметром cast_to. Він робить запит до фізичного 2D рушія, щоб визначити найближчий об’єкт на своєму шляху. Систему пошуку можна гнучко налаштувати під свої потреби. Є можливість

задавання правил для ігнорування конкретних вузлів та їх видів, або для налаштування реакції лише на певний конкретний тип об’єктів.

Основі методи, що використовуються у проекті:

- add_exception (Object node) – додати об’єкт, що буде ігноруватися при запиті 2D рушія на перше зіткнення
- get_collider () Object – повертає перший об’єкт з яким перетинається промінь або null, якщо такого немає
- is_colliding () bool – функція, що перевіряє чи перетинається промінь із одним з дозволених об’єктів
- get_collision_point () Vector2 – повертає точку перетину у двовимірному просторі, якщо такий наявний
- set_collision_mask_bit (int bit, bool value) – дуже потужний інструмент для задавання правил взаємодії з вузлами. Raycast2D реагуватиме лише на об’єкти, що мають такі ж рівні колізії, як задані у параметрі collision_mask даного вузла. Це дозволяє зручно налаштовувати правила взаємодії, коли в грі присутня велика кількість об’єктів різних типів.

Крім цього даний вузол містить ще три важливих параметри:

- enabled bool – відповідає за стан променю, якщо виставлене значення false – вузол не реагуватиме на жодні колізії
- collide_with_areas bool – дозволяє або забороняє перетин з вузлами Area2D
- Collide_with_bodies bool – дозволяє або забороняє перетин з фізичними об’єктами

Дуже потужний вузол, що може бути використаний для реалізації лінії зору штучного інтелекту, обрахування відстані та напрямку до фізичних тіл та для вирішення багатьох інших завдань.

Area2D

Вузол, що являє собою двовимірну область у просторі. Він виявляє перетин з вузлами типу CollisionObject2D та повідомляє про їх вхід або вихід зі своєї зони. Також надає можливість змінювати локальні параметри фізики (змінювати силу тяжіння та демпфірування) [7]

Сигнали та методи, що використовуються в проекті:

- set_collision_mask_bit (int bit, bool value) – метод, що виставляє для даного вузла вказаний рівень колізії
- area_entered (Area2D area) – сигнал, що повідомляє про входження іншого вузла типу Area2D
- area_exited (Area2D area) – сигнал, що повідомляє про вихід з зони перетину іншого вузла типу Area2D
- body_entered (Node body) - сигнал, що повідомляє про входження фізичного об’єкту
- body_exited (Node body) - сигнал, що повідомляє про вихід з зони перетину фізичного об’єкту

За допомогою даного вузла можна, наприклад, створити поле зору штучного інтелекту, або тригер певних подій при входженні інших вузлів.

CollisionShape2D

Інструмент для створення різних форм (фігур) зіткнення, не є самостійним об’єктом і може використовуватися як частина фізичного тіла.

StaticBody2D

Вузол для симуляції статичного тіла в фізичному двовимірному рушії. Ідеально підходить для реалізації об’єктів у фізичному середовищі, наприклад стін або бар’єрів. Крім того, для статичного тіла може бути встановлена постійна лінійна або кутова швидкість, що матиме вплив на зіткнення з іншими тілами (це дозволяє реалізувати, наприклад, стрічку конвеєра) [7].

KinematicBody2D

Вузол, спеціально розрахований на керування гравцем. Для усіх інших типів вершин, вони він нічим не відрізняється від StaticBody2D, однак фізичний рушій автоматично обчислює їх лінійну та кутову швидкості, що дозволяє реалізувати рухливе тіло. Також даний вузол має API для переміщення об'єктів з урахуванням зіткнень (методи move_and_collide та move_and_slide). Через це він дуже зручний для реалізації ігрових персонажів, що стикаються зі світом, однак для цього не потрібна складна фізика.

Основний метод, що використовуються у проекті – move_and_slide. Він пересуває тіло вздовж заданого вектора. При зіткненні з іншим об'єктом, батьківський вузол проковзне по ньому, а не просто зупиниться. Якщо інше тіло має тип KinematicBody2D або RigidBody2D, це зіткнення вплине на його рух згідно з заданими в проекті законами фізики.

RigidBody2D

Вузол, яким повністю керує фізичний 2D рушій. Розробник не має прямої можливості змінювати положення та швидкість цього тіла. Натомість до нього необхідно застосовувати сили (сила тяжіння, імпульси тощо), а фізичне моделювання обчислює отриманий рух на основі його маси, тертя та інших фізичних властивостей [7].

Основі методи, що використовуються у проекті:

- add_force (Vector2 offset, Vector2 force) - додає постійну позиційну силу до тіла.
- add_torque (float torque) – додає постійну силу обертання.
- apply_impulse (Vector2 offset, Vector2 impulse) – застосовує позиційний імпульс, який не залежить від часу, до даного тіла. З цієї причини його слід використовувати лише при моделюванні одноразових фізичних впливів.

Timer

Вузол, шо реалізує таймер зворотного відліку. Відраховує заданий інтервал і видає сигнал при досягненні 0. Можна встановити режим повтору або режим "один запуск".

AnimationPlayer

Вузол, що реалізує програвач анімацій. Він містить набір анімацій із можливістю задавати переходи та їх тривалість. Крім того, анімації можна відтворювати та змішувати на різних каналах.

AnimationTree

AnimationTree – вузол для створення складних переходів між анімаціями в AnimationPlayer.

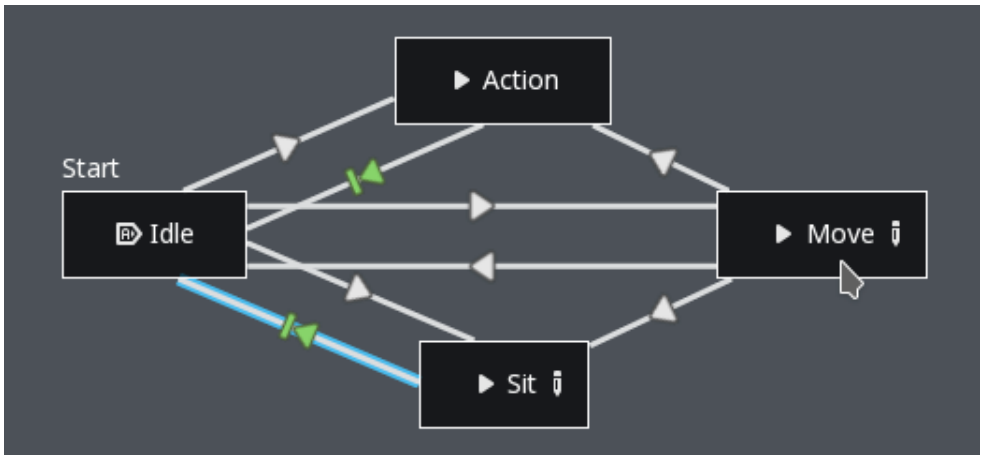


Рисунок 3.4 – Графічний редактор AnimationTree

Control

Усі вузли інтерфейсу користувача успадковуються від цієї вершини. Параметри Anchors та Margins пристосовують його положення та розмір відносно батьківського вузла. Control містить обмежуючий прямокутник, який визначає його розширення, положення якоря відносно його батьківського елемента керування або поточного вікна перегляду, а також поля, що представляють собою зміщення якоря. Поля автоматично оновлюються, коли вузол, будь-який з його батьків або розмір екрана змінюються.

Container

Базовий вузол для контейнерів. Містить в собі інші елементи управління і автоматично їх упорядковує вказаним чином.

ProgressBar

Відображає шкалу заповнення, можна використовувати в якості індикатора прогресу або здоров’я гравця. Однак даний вузол обмежений в плані модифікацій візуального стилю. Розробник має можливість лише обрати колір заповнення та розмір шкали.

TextureProgress

TextureProgress працює як ProgressBar, але використовує до 3 текстур замість простого ресурсу заповнення. Його можна використовувати для створення горизонтальних, вертикальних і радіальних смуг прогресу.

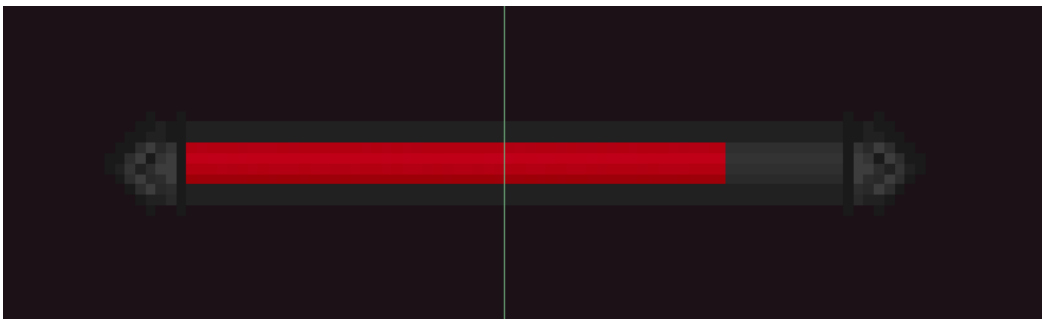


Рисунок 3.5 – Шкала здоров’я, реалізована за допомогою TextureProgress

Label

Вузол, що відповідає за графічне відображення звичайного тексту на екрані. Він надає змогу контролювати горизонтальне та вертикальне вирівнювання, а також може загортати текст всередині обмежуючого прямокутника. Він не підтримує жирне, курсивне та інше форматування. Для цього слід натомість використовувати RichTextLabel.

Particles2D

Генератор двовимірних частинок. Активно застосовується для створення різноманітних систем та ефектів. Має в собі випромінювач, який генерує задану кількість частинок із вказаними параметрами (лінійна та кутова швидкості, розмір, напрямок, випадковість, тощо). Підтримує можливість використання текстури як ресурсу для відображення кожної конкретної частинки. Також може використовувати шейдери для накладання додаткових графічних ефектів.

Генерація початкового поля астероїдів

Один із найпростіших способів згенерувати велику кількість випадкових об’єктів – використовувати шум. На даний момент у відкритому доступі є декілька основних варіантів:

- Шум Перліна (англ. Perlin Noise) – найпростіший варіант. Цей шум був розроблений Кеном Перліном в 1983 році та має декілька значних недоліків. Є високий шанс отримати помітні візуальні артефакти, при цьому даний алгоритм працює доволі повільно
- Симплекс шум (англ. Simplex Noise) – розроблений Кеном Перліном в 2001 році як спроба усунення недоліків шуму Перліна. Потужне та швидке рішення, однак використання тривимірного симплекс шуму захищене патентом.

- Відкритий симплекс шум (англ. Open Simplex Noise) – був розроблений KDotJPG з однією простою метою – створити сучасну та безкоштовну версію симплекс шуму, відносно швидку і без спотворень.

В даному проекті, для генерації початкового стану, положення та концентрації астероїдів, за основу було взято останній варіант – відкритий симплекс шум. Він представлений у обраному ігровому рушії в якості простого інструменту та має наступні параметри для налаштування:

- float lacunarity – визначає різницю в періоді між октавами
- int octaves – задає кількість шарів, відібраних для отримання фрактального шуму. Вище значення даного параметра призводить до вищої детальності шуму, однак вимагає більше часу для генерування (максимальне дозволене значення - 9)
- float period – період базової октави. Нижче значення призводить до більш високої частоти шуму (більша зміна для однакової відстані)
- float persistence – коефіцієнт внеску різних октав. Значення стійкості 1 означає, що всі октави мають однаковий внесок, значення 0,5 означає, що кожна октава вносить вдвічі менше змін, ніж попередня.
- int seed – джерело, яке використовується для генерування випадкових значень, різні джерела генерують різні карти шуму

Приклад використання:

```
var noise = OpenSimplexNoise.new()

# Configure
noise.seed = randi()
noise.octaves = 4
noise.period = 20.0
noise.persistence = 0.8

# Sample
print("Values:")
print(noise.get_noise_2d(1.0, 1.0))
print(noise.get_noise_3d(0.5, 3.0, 15.0))
print(noise.get_noise_4d(0.5, 1.9, 4.7, 0.0))
```

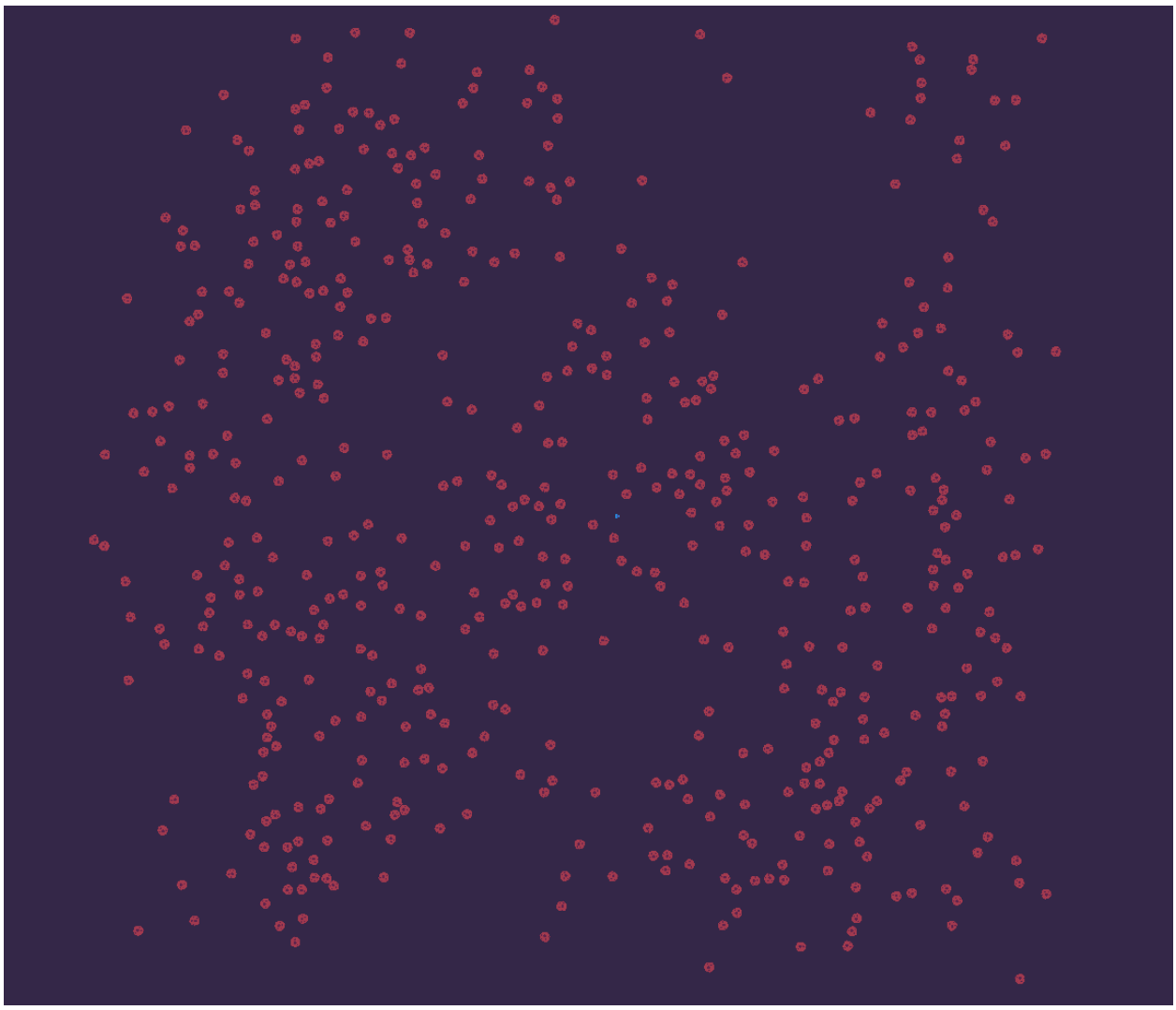


Рисунок 3.6 – Згенероване поле астероїдів

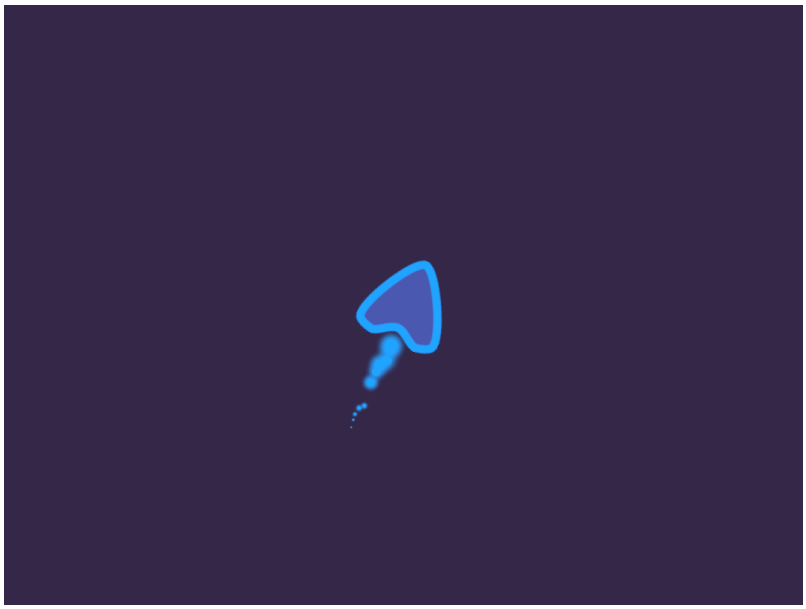


Рисунок 3.7 – Гравець в Astroboii

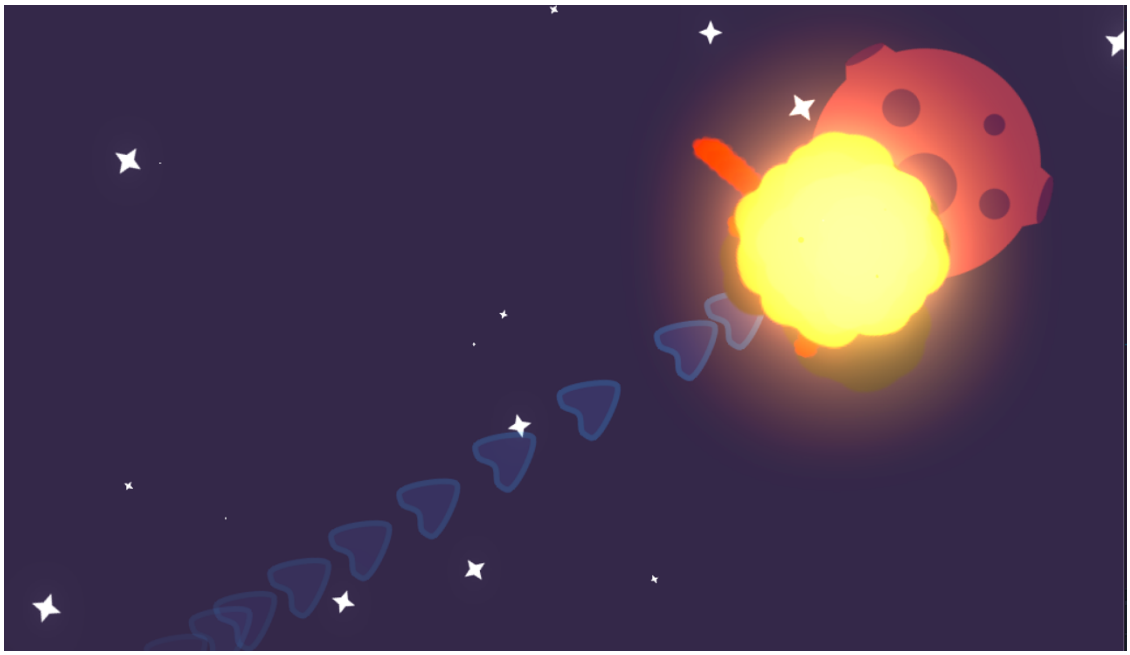


Рисунок 3.8 – Ефект вибуху в Astroboii

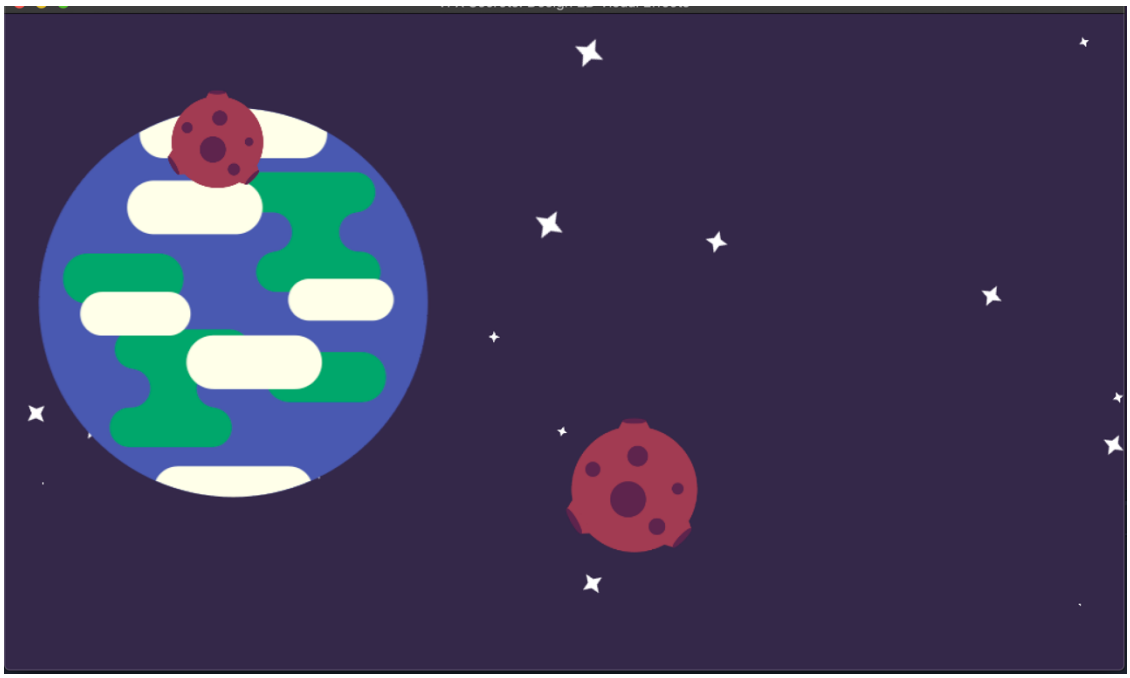


Рисунок 3.8 – Ігрове поле з додатковими декораціями

ВИСНОВОК ДО РОЗДІЛУ 3

У даному розділі було описано загальну архітектуру проекту та взаємодію різноманітних ігрових компонентів між собою. Розглянуто основні інструменти, що надаються обраним ігровим рушієм, необхідні для реалізації проекту із прикладами їх безпосереднього використання. Було розроблено програмну реалізацію, яка задовольняє поставлені вимоги. Внаслідок гнучкої архітектури проекту, а також зручних та потужних інструментів рушія, можна доповнювати, модифікувати та розширювати створену гру без за короткі проміжки часу. Також протестована можливість збірки та запуску проекту на різних операційних системах та апаратних платформах.

ВИСНОВКИ

Даний дипломний проект присвячений розробці комп’ютерної гри в жанрі Аркада. Для формування ігрових механік та базових ідей було проведено огляд існуючих відомих рішень, що відносяться до даного жанру і значною мірою вплинули на його розвиток та популяризацію.

В ході роботи було проведено аналіз актуальності даного роду занять станом на теперішній час. Було надано загальний опис предметної області, визначені основні елементи, механіки та графічний стиль проекту, сформований перелік основних вимог до кінцевої реалізації. Перед початком імплементації було проведено аналіз існуючих ігрових рушіїв у відкритому доступі та обрано найбільш підходящий варіант.

Створена відеогра має простий користувацький інтерфейс та дуже базові ігрові механіки. Оскільки вона обмежена двовимірним простором та має не складну графічну складову, то кінцеві вимоги до апаратної системи користувача є дуже не високими. Крім цього, завдяки можливості обраного ігрового рушія збирати проект для роботи на різних популярних операційних системах, портувати дану гру під різні платформи дуже просто. Особливо це полегшує випуск кінцевого продукту для системи Linux, адже у багатьох ігрових рушіях є проблеми з компіляцією проекту під дану платформу, або ця операція вимагає великої кількості зусиль та часу ігрових розробників.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Free 2D Game Engine comparison [Електронний ресурс] – Режим доступу: https://careerkarma.com/blog/2d-game-engines/
2. Behavior Tree [Електронний ресурс] – Режим доступу: http://web.archive.org/web/20140402204854/http://www.altdevblogaday.com/2011/02/24/introduction-to-behavior-trees/
3. RPC: Protocol Specification [Електронний ресурс] Режим доступу: https://docs.freebsd.org/44doc/psd/26.rpcrfc/paper.pdf
4. Unity games [Електронний ресурс] – Режим доступу: https://unity3d.com/ru/games-made-with-unity
5. ASP.NET MVC Overview [Електронний ресурс]. – Microsoft, 2016. – Режим доступу: https://msdn.microsoft.com/enus/library/dd381412(v=vs.108).aspx.
6. Design Patterns MVC Pattern [Електронний ресурс]. – TutorialsPoint, 2016. – Режим доступу: www.tutorialspoint.com/design_pattern/mvc_pattern.htm.
7. Godot docs [Електронний ресурс] – Режим доступу: https://docs.godotengine.org/en/stable/classes/class_node.html
8. Godot getting started [Електронний ресурс] – Режим доступу: https://docs.godotengine.org/en/stable/getting_started/step_by_step/scenes_and_nodes.html
9. Behavior Tree [Електронний ресурс] – Режим доступу: https://www.gamasutra.com/view/feature/130663/gdc_2005_proceeding_handling_.php
10. Pushdown Automaton [Електронний ресурс] – Режим доступу: https://gameprogrammingpatterns.com/state.html#pushdown-automata
11. State Machine [Електронний ресурс] – Режим доступу: https://gameprogrammingpatterns.com/state.html#pushdown-automata

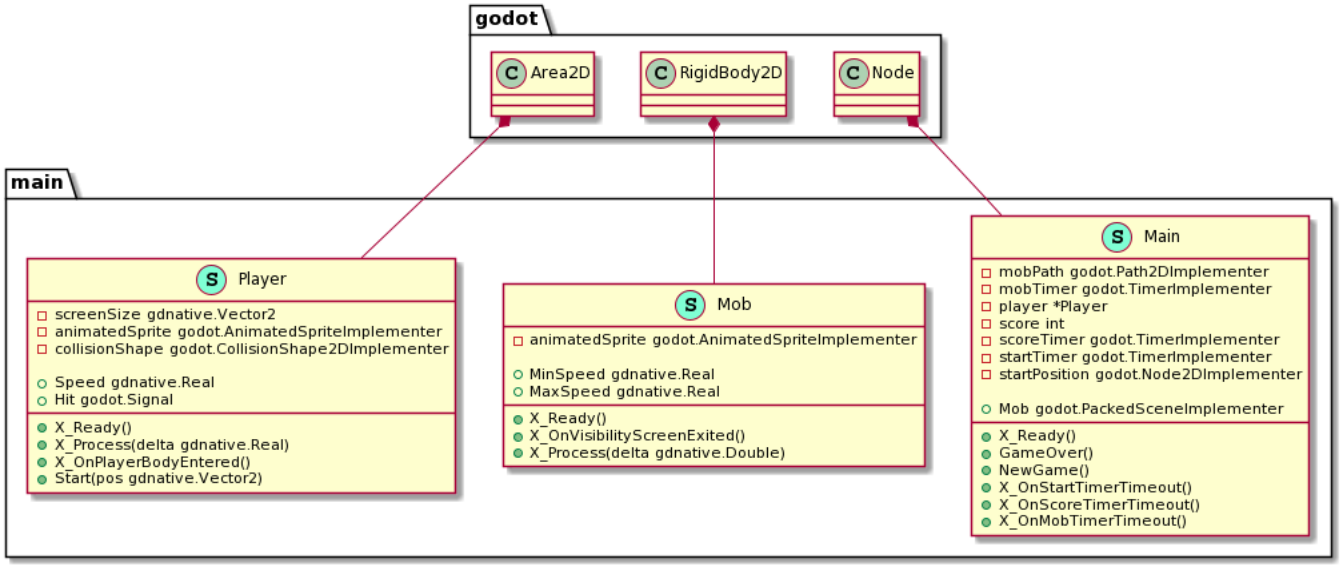
12. Бартіш М. Я. Теорія ігор / Бартіш М. Я., Роман Л. Л. – Львів: Видавничий центр ЛНУ, 2005. – 120 с.
13. The Art of Game Design: A Book of Lenses, Second Edition by Jesse Schell, 2020 – 60c.
14. Theory of Fun for Game Design by Raph Koster, 2013 – 96c.
15. Game Programming Patterns 1st Edition by Robert Nystrom, 2014 – 30c.
16. Learning C++ by Creating Games with UE4 by William Sherif. - 2015.
17. Unreal Networking Guide by Zach Metcalf. – 2019.
18. Mathematics for 3D Game Programming and Computer Graphics, Third Edition. – 2012
19. Georgios N. Yannakakis. Game AI Revisited / Georgios N. Yannakakis. – Copenhagen, 2012. – 15c.
20. Brian Schwab AI Game Engine Programming - Course Technology Cengage Learning, 2009. - 710 c.
21. OpenSimplexNoise [Електронний ресурс] - Режим доступу: https://docs.godotengine.org/uk/stable/classes/class_opensimplexnoise.html
22. Eric Eaton. Who speaks for AI? / Eric Eaton, Tom Dietterich, Maria Gini. // AI Matters. - 2015. - No2. - –°. 4-14.

Додаток 1

до дипломного проекту

на тему: «Комп’ютерна гра в жанрі Аркада»

Київ – 2020 року



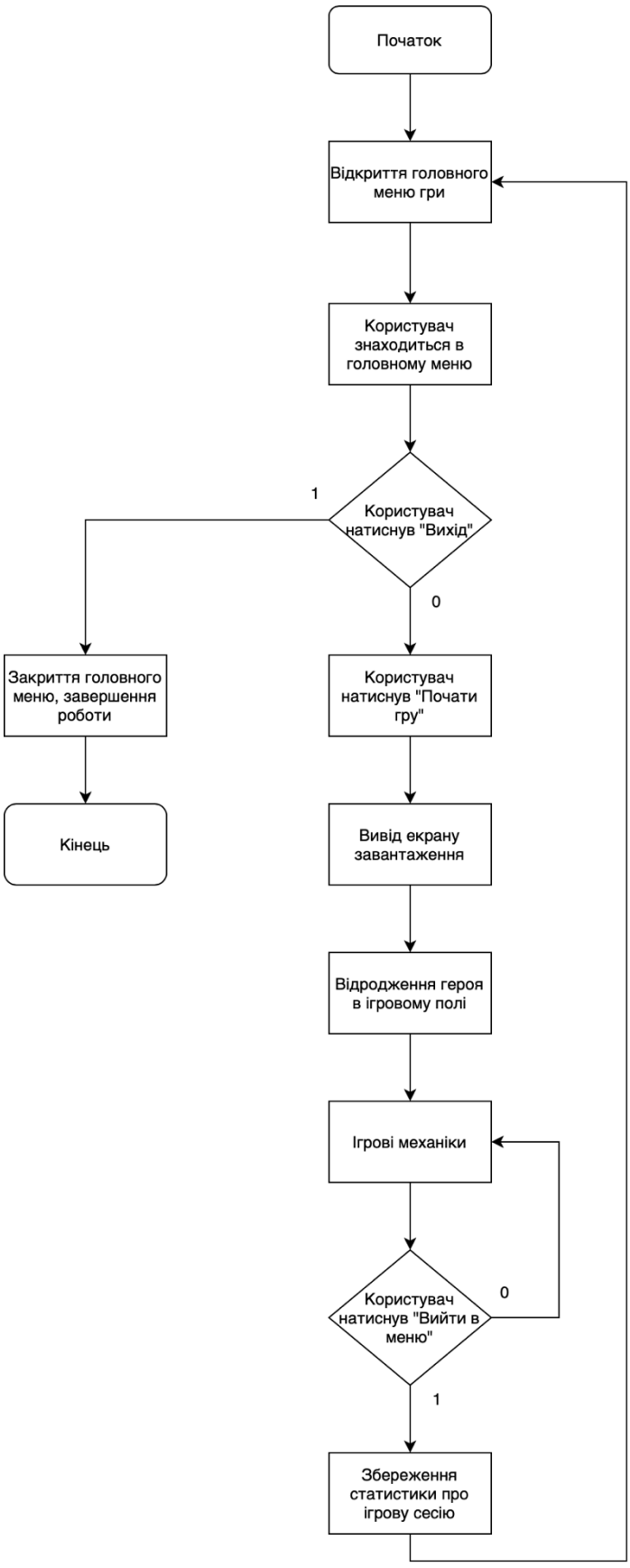
					ІАЛЦ. 467800.004 Д1			
Зм.	Арк.	№ докум.	Підпис	Дата				
Розробив		Черпатюк А.С.			Комп'ютерна гра в жанрі Аркада	Літ.	Аркуш	Аркушів
Перевір.		Верба О.А.			Схема класів		1	1
						НТУУ "КПІ ім. Ігоря Сікорського", ФІОТ, ІО-		
Н. контр.		Сімоненко В.П.						
Затверд.								

Додаток 2

до дипломного проекту

на тему: «Комп’ютерна гра в жанрі Аркада»

Київ – 2020 року



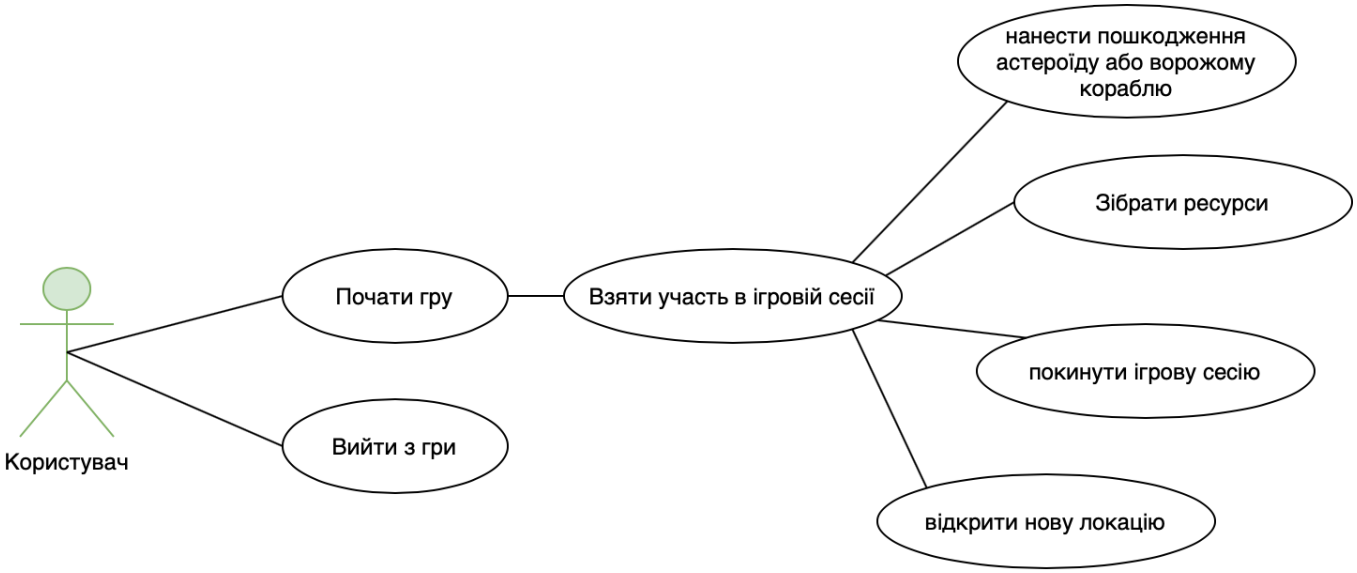
					ІАЛЦ. 467800.005 Д2		
Зм.	Арк.	№ докум.	Підпис	Дата			
Розробив		Черпатюк А.С.			Комп'ютерна гра в жанрі Аркада	Літ.	Аркуш 1 / Аркушів 1
Перевір.		Верба О.А.			Алгоритм роботи програми		НТУУ "КПІ ім. Ігоря Сікорського", ФІОТ, ІО-
Н. контр.		Сімоненко В.П.					
Затверд.							

Додаток 3

до дипломного проекту

на тему: «Комп’ютерна гра в жанрі Аркада»

Київ – 2020 року



					ІАЛЦ. 467800.006 Д3			
Зм.	Арк.	№ докум.	Підпис	Дата				
Розробив		Черпатюк А.С.			Комп'ютерна гра в жанрі Аркада	Літ.	Аркуш	Аркушів
Перевір.		Верба О.А.			Схема взаємодії з користувачем		1	1
						НТУУ "КПІ ім. Ігоря Сікорського", ФІОТ, ІО-		
Н. контр.		Сімоненко В.П.						
Затверд.								

Додаток 4

до дипломного проекту

на тему: «Комп’ютерна гра в жанрі Аркада»

Київ – 2020 року

ТЕКСТ ПРОГРАМИ

```
extends Node2D

onready var camera = $Camera2D

onready var zoom_value = 0.1
onready var zoom_delta = Vector2(zoom_value, zoom_value)
onready var zoom_lbound = Vector2(0.25, 0.25) # lower bound
onready var zoom_hbound = Vector2(30, 30)     # higher bound

onready var asteroid_scene = preload("res://Asteroid.tscn")

var noise = OpenSimplexNoise.new()

func _ready():
    randomize()
    noise.seed = randi()
    noise.octaves = 8
    noise.period = 10
    noise.persistence = 0.7

    for i in range(30):
         for j in range(30):
              if noise.get_noise_2d(i, j) < 0:
                   var asteroid = asteroid_scene.instance()
                   asteroid.position = (Vector2(i, j) * 600) -
Vector2(15 * 600, 15 * 600)
                   get_tree().get_root().call_deferred("add_child",
asteroid)

func _unhandled_input(event):
    var zoom = camera.get_zoom()
    if event is InputEventMouseButton:
         if event.button_index == BUTTON_WHEEL_UP:
              if zoom <= zoom_hbound:
                   zoom += zoom_delta
         if event.button_index == BUTTON_WHEEL_DOWN:
              if(zoom - zoom_delta > zoom_lbound):
                   zoom -= zoom_delta
         camera.set_zoom(zoom)

    if event is InputEventPanGesture:
         var delta = event.delta.y * 0.1
         zoom += Vector2(delta, delta)
         if zoom > zoom_lbound and zoom < zoom_hbound:
              camera.set_zoom(zoom)
```

```
extends KinematicBody2D

const TurnSpeed := 160

const MoveSpeed := 500
const Acc = 1
const Dec = 0.1

var motion = Vector2.ZERO

func _process(delta: float):
    if Input.is_action_pressed("move_left"):
          rotation_degrees -= TurnSpeed * delta
    if Input.is_action_pressed("move_right"):
          rotation_degrees += TurnSpeed * delta

    var move_dir = Vector2(1, 0).rotated(rotation)

    if Input.is_action_pressed("move_up"):
          $Particles2D.emitting = true
          motion = motion.linear_interpolate(move_dir, Acc * delta)
    else:
          $Particles2D.emitting = false
          motion = motion.linear_interpolate(Vector2.ZERO, Dec * delta)

    position += motion * MoveSpeed * delta

extends RigidBody2D

var speed = 250
var velocity = Vector2()

# Called when the node enters the scene tree for the first time.
func _ready():
    randomize()
    linear_velocity = Vector2(rand_range(40, 90),
0).rotated(rand_range(0, 2*PI))
    angular_velocity = rand_range(-1.5, 1.5)

    mass = 9
    gravity_scale = 0.0

    bounce = 0.8

    linear_damp  = 0
    angular_damp = 0
```

```
extends RayCast2D

export var cast_speed := 7000.0
export var max_length := 1400
export var growth_time := 0.1

var is_casting := false setget set_is_casting

onready var fill := $FillLine2D
onready var tween := $Tween
onready var casting_particles := $CastingParticles2D
onready var collision_particles := $CollisionParticles2D
onready var beam_particles := $BeamParticles2D

onready var line_width: float = fill.width


func _ready() -> void:
    set_physics_process(false)
    fill.points[1] = Vector2.ZERO


func _physics_process(delta: float) -> void:
    cast_to = (cast_to + Vector2.RIGHT * cast_speed *
delta).clamped(max_length)
    cast_beam()


func set_is_casting(cast: bool) -> void:
    is_casting = cast

    if is_casting:
         cast_to = Vector2.ZERO
         fill.points[1] = cast_to
         appear()
    else:
         collision_particles.emitting = false
         disappear()

    set_physics_process(is_casting)
    beam_particles.emitting = is_casting
    casting_particles.emitting = is_casting


func cast_beam() -> void:
    var cast_point := cast_to

    force_raycast_update()
    collision_particles.emitting = is_colliding()

    if is_colliding():
         cast_point = to_local(get_collision_point())
         collision_particles.process_material.direction = Vector3(
              get_collision_normal().x, get_collision_normal().y, 0
         )
         collision_particles.position = cast_point
```

```
    fill.points[1] = cast_point
    beam_particles.position = cast_point * 0.5
    beam_particles.process_material.emission_box_extents.x =
cast_point.length() * 0.5


func appear() -> void:
    if tween.is_active():
         tween.stop_all()
    tween.interpolate_property(fill, "width", 0, line_width,
growth_time * 2)
    tween.start()


func disappear() -> void:
    if tween.is_active():
         tween.stop_all()
    tween.interpolate_property(fill, "width", fill.width, 0,
growth_time)
    tween.start()



# Draws a 2D trail using Godot's `Line2D`.
#
# Instantiate `Trail2D` as a child of a moving node to use it. To
control the color, width curve,
# texture, or trail width, use parameters from the `Line2D` class.
tool

class_name Trail2D
extends Line2D

export var is_emitting := false setget set_emitting

# Distance in pixels between vertices. A higher resolution leads to
more details.
export var resolution := 5.0



# Life of each point in seconds before it is deleted.
export var lifetime := 0.5
# Maximum number of points allowed on the curve.
export var max_points := 100

# Optional path to the target node to follow. If not set, the instance
follows its parent.
export var target_path: NodePath

var _points_creation_time := []
var _last_point := Vector2.ZERO
var _clock := 0.0
var _offset := 0.0
```

```
onready var target: Node2D = get_node_or_null(target_path)


func _ready() -> void:
    if not target:
        target = get_parent() as Node2D

    if Engine.editor_hint:
        set_process(false)
        return

    _offset = position.length()
    set_as_toplevel(true)
    clear_points()
    position = Vector2.ZERO
    _last_point = to_local(target.global_position) +
calculate_offset()


func _get_configuration_warning() -> String:
    var warning := "Missing Target node: assign a Node that extends
Node2D in the Target Path or make the Trail a child of a parent that
extends Node2D"
    if target:
        warning = ""
    return warning


func _process(delta: float) -> void:
    _clock += delta
    remove_older()

    if not is_emitting:
        return

    # Adding new points if necessary.
    var desired_point := to_local(target.global_position)
    var distance: float = _last_point.distance_to(desired_point)
    if distance > resolution:
        add_timed_point(desired_point, _clock)


# Creates a new point and stores its creation time.
func add_timed_point(point: Vector2, time: float) -> void:
    add_point(point + calculate_offset())
    _points_creation_time.append(time)
    _last_point = point
    if get_point_count() > max_points:
        remove_first_point()


# Calculates the offset of the trail from its target.
func calculate_offset() -> Vector2:
    return -polar2cartesian(1.0, target.rotation) * _offset
```

```
# Removes the first point in the line and the corresponding time.
func remove_first_point() -> void:
    if get_point_count() > 1:
        remove_point(0)
    _points_creation_time.pop_front()


# Remove points older than `lifetime`.
func remove_older() -> void:
    for creation_time in _points_creation_time:
        var delta = _clock - creation_time
        if delta > lifetime:
            remove_first_point()
        # Points in `_points_creation_time` are ordered from oldest
to newest so as soon as a point
        # isn't older than `lifetime`, we know all remaining points
should stay as well.
        else:
            break


func set_emitting(emitting: bool) -> void:
    is_emitting = emitting
    if Engine.editor_hint:
        return

    if not is_inside_tree():
        yield(self, "ready")

    if is_emitting:
        clear_points()
        _points_creation_time.clear()
        _last_point = to_local(target.global_position) +
calculate_offset()

package main

import (
    gd "github.com/shadowapex/godot-go/gdnative"
    "github.com/shadowapex/godot-go/godot"
    "log"
    "math/rand"
)

func main() {
}

// Init will be called when this library is loaded by Godot.
func init() {
    // Set up logging to log to Godot.
    log.SetFlags(log.LstdFlags | log.Lshortfile)
    log.SetOutput(godot.Log)

    // AutoRegister our Player and Mob classes.
    godot.AutoRegister(NewMain)
    godot.AutoRegister(NewPlayer)
```

```
    godot.AutoRegister(NewMob)
}

func NewMain() godot.Class {
    m := &Main{}

    return m
}

// Main is a structure for the main game script.
type Main struct {
    godot.Node
    Mob           godot.PackedSceneImplementer `hint:"ResourceType"
usage:"Default"`
    mobPath       godot.Path2DImplementer
    mobTimer      godot.TimerImplementer
    player        *Player
    score         int
    scoreTimer    godot.TimerImplementer
    startTimer    godot.TimerImplementer
    startPosition godot.Node2DImplementer
}

// X_Ready will be called as soon as the main node enters the scene.
func (m *Main) X_Ready() {
    log.Println("X_Ready called!")
    log.Println("Registry:", godot.InstanceRegistry)

    // Get our score and mob timers
    scoreTimerPath := gd.NewNodePath("ScoreTimer")
    scoreTimerNode := m.GetNode(scoreTimerPath)
    m.scoreTimer = scoreTimerNode.(godot.TimerImplementer)

    startTimerPath := gd.NewNodePath("StartTimer")
    startTimerNode := m.GetNode(startTimerPath)
    m.startTimer = startTimerNode.(godot.TimerImplementer)

    mobTimerPath := gd.NewNodePath("MobTimer")
    mobTimerNode := m.GetNode(mobTimerPath)
    m.mobTimer = mobTimerNode.(godot.TimerImplementer)

    // Get the player
    playerPath := gd.NewNodePath("Player")
    playerNode := m.GetNode(playerPath)
    m.player = playerNode.(*Player)

    // Get the starting position
    startPosPath := gd.NewNodePath("StartPosition")
    startPosNode := m.GetNode(startPosPath)
    m.startPosition = startPosNode.(godot.Node2DImplementer)

    // Get the mob path
    mobPathPath := gd.NewNodePath("MobPath")
    mobPathNode := m.GetNode(mobPathPath)
    m.mobPath = mobPathNode.(godot.Path2DImplementer)
```

```
    // Start the game
    m.NewGame()
}

func (m *Main) GameOver() {
    log.Println("Game over!")
    m.scoreTimer.Stop()
    m.mobTimer.Stop()
}

func (m *Main) NewGame() {
    log.Println("New Game")
    m.score = 0
    m.player.Start(m.startPosition.GetPosition())
    m.startTimer.Start()
}

func (m *Main) X_OnStartTimerTimeout() {
    log.Println("Start timer timeout")
    m.mobTimer.Start()
    m.scoreTimer.Start()
}

func (m *Main) X_OnScoreTimerTimeout() {
    log.Println("Score timer timeout")
    m.score += 1
}

func (m *Main) X_OnMobTimerTimeout() {
    log.Println("Mob timer timeout")
    mobSpawnLocationPath := gd.NewNodePath("MobSpawnLocation")
    mobSpawnLocation :=
(m.mobPath.GetNode(mobSpawnLocationPath)).(godot.PathFollow2DImplement
er)
    mobSpawnLocation.SetOffset(gd.Real(rand.Int()))

    // Create a mob instance and add it to the scene
    if m.Mob.CanInstance() {
        mob :=
m.Mob.Instance(gd.Int(godot.PackedSceneGenEditStateDisabled))
        m.AddChild(mob, false)
    }
}

package main

import (
    gd "github.com/shadowapex/godot-go/gdnative"
    "github.com/shadowapex/godot-go/godot"
    "log"
    "math/rand"
)

func NewMob() godot.Class {
    mob := &Mob{}
```

```
    return mob
}

// Mob is a structure for enemy mobs.
type Mob struct {
    godot.RigidBody2D
    MinSpeed       gd.Real
    MaxSpeed       gd.Real
    animatedSprite godot.AnimatedSpriteImplementer
}

// X_Ready will be called as soon as the mob enters the scene.
func (m *Mob) X_Ready() {
    log.Println("X_Ready called!")

    // Get the AnimatedSprite child node.
    animatedSpritePath := gd.NewNodePath("AnimatedSprite")
    animatedSpriteNode := m.GetNode(animatedSpritePath)
    m.animatedSprite =
animatedSpriteNode.(godot.AnimatedSpriteImplementer)

    // Set up different mob types
    mobTypes := []gd.String{"walk", "swim", "fly"}

    // Randomly select a mob type
    m.animatedSprite.SetAnimation(mobTypes[rand.Int()%len(mobTypes)])
}

func (m *Mob) X_OnVisibilityScreenExited() {
    m.QueueFree()
}

// X_Process will be called every frame.
func (m *Mob) X_Process(delta gd.Double) {
}

package main

import (
    gd "github.com/shadowapex/godot-go/gdnative"
    "github.com/shadowapex/godot-go/godot"
    "log"
)

// NewPlayer is a player constructor that we will register with Godot.
func NewPlayer() godot.Class {
    player := &Player{
         Hit: godot.Signal{
              Name:        "hit",
              Args:        []godot.SignalArg{},
              DefaultArgs: []godot.SignalDefaultArg{},
         },
    }

    return player
}
```

```
// Player is a structure for the player.
type Player struct {
    godot.Area2D
    Speed          gd.Real `hint_string:"The speed of the player"`
    Hit            godot.Signal
    screenSize     gd.Vector2
    animatedSprite godot.AnimatedSpriteImplementer
    collisionShape godot.CollisionShape2DImplementer
}

// X_Ready will be called as soon as the player enters the scene.
func (p *Player) X_Ready() {
    log.Println("X_Ready called!")

    // Get the AnimatedSprite child node.
    log.Println("Getting animated sprite...")
    animatedSpritePath := gd.NewNodePath("AnimatedSprite")
    animatedSpriteNode := p.GetNode(animatedSpritePath)
    log.Println("Got animated sprite with ID:",
animatedSpriteNode.GetBaseObject().ID())
    p.animatedSprite =
animatedSpriteNode.(godot.AnimatedSpriteImplementer)

    // Get the collision shape child node.
    collisionShapePath := gd.NewNodePath("CollisionShape2D")
    collisionShapeNode := p.GetNode(collisionShapePath)
    p.collisionShape =
collisionShapeNode.(godot.CollisionShape2DImplementer)

    // Get the viewport size
    viewportRect := p.GetViewportRect()
    p.screenSize = viewportRect.GetSize()
}

// X_Process will be called every frame.
func (p *Player) X_Process(delta gd.Real) {
    velocity := gd.NewVector2(0, 0)

    if godot.Input.IsActionPressed("ui_right") {
         velocity.SetX(velocity.GetX() + 1)
    }
    if godot.Input.IsActionPressed("ui_left") {
         velocity.SetX(velocity.GetX() - 1)
    }
    if godot.Input.IsActionPressed("ui_down") {
         velocity.SetY(velocity.GetY() + 1)
    }
    if godot.Input.IsActionPressed("ui_up") {
         velocity.SetY(velocity.GetY() - 1)
    }

    if velocity.Length() > 0 {
         normal := velocity.Normalized()
         velocity = normal.OperatorMultiplyScalar(p.Speed)
    } else {
```

```
            p.animatedSprite.Stop()
        }

        // Set the position based on velocity
        position := p.GetPosition()
        newPosition :=
    position.OperatorAdd(velocity.OperatorMultiplyScalar(delta))

        // Clamp our player's position to the size of the screen.
        newPosition.SetX(godot.Clamp(newPosition.GetX(), 0,
    p.screenSize.GetX()))
        newPosition.SetY(godot.Clamp(newPosition.GetY(), 0,
    p.screenSize.GetY()))
        p.SetPosition(newPosition)

        // Flip the sprite if we're moving left or down, since we only
    have sprites
        // for right and up.
        if velocity.GetX() != 0 {
            p.animatedSprite.SetAnimation("right")
            p.animatedSprite.SetFlipV(false)
            p.animatedSprite.SetFlipH(velocity.GetX() < 0)
        } else if velocity.GetY() != 0 {
            p.animatedSprite.SetAnimation("up")
            p.animatedSprite.SetFlipV(velocity.GetY() > 0)
        }
    }

    func (p *Player) X_OnPlayerBodyEntered() {
        log.Println("Player body was entered!")
        p.Hide()
        p.EmitSignal("hit")
        p.collisionShape.SetDisabled(true)
    }

    // Start will be called when we need to reset our player's position.
    func (p *Player) Start(pos gd.Vector2) {
        p.SetPosition(pos)
        p.Show()
        p.collisionShape.SetDisabled(false)
    }
```